Abstract Interpretation using Attribute Grammars

Mads Rosendahl
Computer Laboratory
University of Cambridge
Cambridge CB2 3QG
England

Extended Abstract

Abstract

This paper deals with the correctness proofs of attribute grammars using methods from abstract interpretation. The technique will be described by defining a live-variable analysis for a small flow-chart language and proving it correct with respect to a continuation style semantics. The proof technique is based on fixpoint induction and introduces an extended class of attribute grammars as to express a standard semantics.

1 Introduction

The proofs of data-flow analysis algorithms has been formalised in the framework of abstract interpretation. In this paper we show how a data-flow analysis can be expressed as an attribute grammar and proved correct using methods from abstract interpretation. It gives a method to prove an attribute grammar correct by relating it to a semantics but it also provides an implementation strategy for abstract interpretations.

The concept of dependencies in attribute grammars provides a simple method to describe the direction of flow in a semantics. Furthermore, attribute grammars provide a useful tool for compiler writing and if abstract interpretation algorithms are available in the same framework it could be easier to use the results for highly optimising compilers.

An abstract interpretation must be proved correct by establishing a relationship with a standard semantics. To be able to do that in the framework of attribute grammars it is necessary to introduce a more general class of attribute grammars where it is possible to specify a useful semantics for a language. The abstract interpretation itself can often be in a more restricted class of attribute grammars where efficient evaluators are available.

As an example the paper describes a specification and proof of the live-variable analysis. An implementation will be discussed and in fact both the standard and the abstract interpretation can be directly executed. The choice of the live-variable analysis example has been inspired by [Babich & Jazayeri 1978] where it was expressed in a "near"-attribute grammar framework.

1.1 Abstract interpretation

Abstract interpretation is a proof technique for data flow analysis algorithms. Programs are given two different meanings where one is a special meaning which if carefully defined can answer questions about the run-time behaviour of the program. The validity of this interpretation is expressed as a relationship between this special meaning of a program and its standard interpretation.

An abstract interpretation can be seen as a special case of a data-flow analysis where the proof is based on a simple relationship between the domains in the two interpretations. This means that the proof and specification of an abstract interpretation must go hand in hand—it is difficult to specify an interpretation without having a relationship in mind and vice versa.

The proof method was pioneered by [Cousot & Cousot 1977] and they expressed the solution to a number of data-flow analysis problems in the framework of abstract interpretations. A bibliography of works related to abstract interpretation and a number of recent articles was published in [Abramsky & Hankin 1987].

In [Cousot & Cousot 1977] a classification of abstract interpretations according to the direction of flow of information. A *forward* analysis will express the information sent to successors of a term as a function of information sent to it from predecessors. In a *backward* analysis the information for a term is related to its successors.

An abstract interpretation is often expressed in the framework of denotational semantics (see [Gordon 1979], and [Schmidt 1986]) where the meaning is expressed as the fixpoint of an equation system of continuous functions over complete partial orders (cpo's). The proof of an abstract interpretation has traditionally been expressed using abstraction/concretisation functions to relate the cpo's in the two interpretations. A more general approach uses relations and it has been explored in [Mycroft & Jones 1986] and [Nielson & Nielson 1988].

1.2 Attribute grammars

Attribute grammars were introduced by [Knuth 1968] as a method to describe the "meaning" or translation of context-free grammars. A survey with an extensive bibliography can be found in [Deransart, Jourdan & Lorho 1988]. The relationship between attribute grammars and denotational semantics has been discussed in [Mayoh 1981] and [Chirica & Martin 1979] and it is shown that the meaning of an attribute grammar can be given as a domain-theoretic fixpoint. The class of attribute grammars which can be defined in this way is larger than in the original definition. This is mainly because it is possible to give meaning to grammars with circular attribute relations. Evaluaters exists which are able to give meaning to some circular attribute grammars. They will normally be based on a lazy evaluation of attributes [Gallier 1984]. Other works ([Farrow 1986] and [Jones & Simon 1986]) uses fixpoint iteration to give meaning to circular but well-defined attribute grammars. Such extensions are not uncommon—perhaps the most general type of attribute grammars was used by [Knuth 1971] where the attributes could be parse trees which could be given meaning by its own attribute grammar.

Circular attribute grammars can also occur when the attribute grammar framework is used as a semantic language. The "semantic grammars" in [Paulson 1982] and [Paulson 1984] are in general circular but they are not used directly for evaluation.

The correctness of attribute grammars has been considered by [Deransart 1983] and [Courcelle & Deransart 1988]. An attribute grammar can be proved correct with respect to a specification by showing that certain assertions will be true for the attributes.

1.3 Data-flow analysis and attribute grammars

Data-flow analysis methods and global optimisation techniques can be used in optimising compilers to improve the performance of programs. A comprehensive bibliography and description of these methods can be found in [Aho, Sethi & Ullman 1986].

In [Babich & Jazayeri 1978] a data-flow analysis is expressed in the general framework of attribute grammars. It is not a real attribute grammar as attribute rules are circular and the attribute rule for the "goto" statement is specified in a non-standard way. This "method of attributes" gives a conceptually simple algorithm for a well-known problem and the underlying system takes care of the main difficulty in most data flow analysis algorithms: how to combine local information into global information. [Babich & Jazayeri 1978] describes the live-variable analysis. It is discussed frequently in the literature and a comprehensive analysis can be found in [Morel 1984].

The implementation of some other data flow analysis algorithms using attribute grammars are considered in [Wilhelm 1981].

1.4 Outline

The paper is based on an example which should clarify the specification and proof techniques. The next section introduces these methods and in the rest of the paper they will be applied to a small flow-chart language described in section 3. The abstract interpretation approach has three stages: specification, proof, and implementation.

Sections 4-5 define the standard and abstract interpretation of the languages. The relationship between these two interpretations is defined and proved in section 6 and the implementation of the interpretations is discussed in section 7.

2 Domain attribute grammars

In this section we introduce a very general class of attribute grammars, called domain attribute grammars. In a domain attribute grammar the attributes belong to domains (*ie.* complete partial ordered sets) and the meaning of a circular attribute definition is defined as the least fixpoint. This class of attribute grammars is mainly meant for proofs and specification. It is hoped that many data-flow analysis algorithms can actually be expressed in a subclass of non-circular attribute grammars where efficient attribute evaluators are available. In the end of the section we introduce a proof method for attribute grammars based on fixpoint induction and logical relations.

2.1 Domain attribute grammars

Domain attribute grammars are attribute grammars where attribute values belong to domains (cpo's) and where the attribute rules are continuous functions. This is essentially the class of attribute grammars defined by [Chirica & Martin 1979] although they restrict it to flat cpo's.[1] We use a more general definition to allow a standard semantics to be described in the same framework as the abstract interpretation. It is important for our application that both the standard interpretation and the abstract interpretation can be given a meaning as an attribute grammar but it is only important that the abstract interpretation can be evaluated.

The specification of a domain attribute grammar consists of consists of two parts: a scheme and some associated definition. The scheme is essentially an attributed scheme ([Deransart 1983]) where some operator symbols in the attribute rules are defined in the definition part. The difference between the standard interpretation and the abstract interpretation are isolated to the definition part so both use the same attribute scheme.

As a minor notational restriction the number of attributes per terminal has been limited to a synthesised and an inherited one. When attributes belong to domains, the same effect as having several attributes can be achieved by allowing an attribute type to be the cartesian (or lazy) product of domains. In the same way we could have removed inherited attributes and obtained a simple S-attribute grammar ([Deransart, Jourdan & Lorho 1988]) where the attributes are functions of the former inherited attributes to the former synthesised attributes. There seem to be several good reasons to keep

[1] In their words the "..definition .. is more general than the usual Knuthian semantic definition because [it] assigns a meaning even to those trees in which the attribute dependency is circular."

the inherited and synthesised attributes separate in the grammar. Most language specifications will only operate with a limited selection of directions of data-flow. [Cousot & Cousot 1977] distinguishes between forward and backward analysis but mentions that combinations hereof are also possible. The use of inherited and synthesised attributes to describe this direction of data-flow seems to make the specification clearer.

We will not give a full description of the notation used and its semantics. The example in section 4 together with the following points should make it possible to follow the interpretation in section 5 and the proof in section 6.

The notation is similar to Extended Attribute Grammars [Watt & Madsen 1983] and the YACC parser generator [Johnson 1975] and we will adopt a variant of the attributed scheme from [Deransart 1983] for proof technical reasons. For a production

$$\mathbf{A}_0 ::= \mathbf{A}_1 \ \ldots \ \mathbf{A_n}$$

the expression

$$\mathbf{A}_0\,\{\mathbf{e}_0\} ::= \mathbf{A}_1\,\{\mathbf{e}_1\} \ \ldots \ \mathbf{A_n}\,\{\mathbf{e_n}\}$$

defines the attribute rules for the production: $\mathbf{e}_0$ defines the synthesised attribute for $\mathbf{A}_0$ and $\mathbf{e_i}, \mathbf{i} = 1, \ldots, \mathbf{n}$ defines the inherited attribute to $\mathbf{A_i}$. The expressions $\mathbf{e}_0, \ldots, \mathbf{e_n}$ may contain the symbols $\$0, \ldots, \$\mathbf{n}$ where $\$0$ denotes the inherited attribute to $\mathbf{A}_0$ and $\$\mathbf{i}, \mathbf{i} = 1, \ldots, \mathbf{n}$ denotes the synthesised attribute for $\mathbf{A_i}$.

In the notation from [Deransart, Jourdan & Lorho 1988] the above attribute definitions can be expressed as

$$\begin{aligned}
&\mathbf{Syn}^{\mathbf{A}_0} = \mathbf{e}_0\\
&\mathbf{Inh}^{\mathbf{A}_1} = \mathbf{e}_1\\
&\vdots\\
&\mathbf{Inh}^{\mathbf{A_n}} = \mathbf{e_n}\\
&\ \mathbf{where}\\
&\quad \$0 = \mathbf{Inh}^{\mathbf{A}_0}\\
&\quad \$1 = \mathbf{Syn}^{\mathbf{A}_1}\\
&\quad \vdots\\
&\quad \$\mathbf{n} = \mathbf{Syn}^{\mathbf{A_n}}
\end{aligned}$$

The shorter notation has been chosen for clarity as the attribute definitions in this case typically are short expressions.

If $\mathbf{A_i}$ is one of the special terminal symbols: **name** or **number** then $\mathbf{A_i}$ should have no inherited attributes and the synthesised attribute ($\$\mathbf{i}$) is the name or number recognised by the lexical analyser. It is convenient to see these special symbols as non-terminals and the grammar as being extended with an infinite set of productions of the form

$$\mathbf{number}\{1\} ::= \text{"1"}$$
$$\vdots$$

The attribute expressions uses an ML-like syntax. The expressions are build from constants and attribute variables ($\$\mathbf{i}, \mathbf{i} = 0, \ldots, \mathbf{n}$) with constructors and function applications. A few specialities have been added for notational convenience:

- $\mathbf{e}_1, \mathbf{e}_2$ denotes the tuple $(\mathbf{e}_1, \mathbf{e}_2)$
- $\mathbf{e}.1$ selects the first element in a tuple and $\mathbf{e}.2$ selects the second element.
- $\mathbf{f}[\mathbf{x} \mapsto \mathbf{v}]$ is a postfix notation for an updated function:

$$\mathbf{f}[\mathbf{x} \mapsto \mathbf{v}] = \lambda\,\mathbf{y}.\ \mathbf{if}\ \mathbf{y} = \mathbf{x}\ \mathbf{then}\ \mathbf{v}\ \mathbf{else}\ \mathbf{f}(\mathbf{y})$$

2.2 Fixpoint induction

In the attribute rules we will leave some operator symbols unspecified for later definition. These symbols will be given two different interpretations where the first will define the standard interpretation and the second the abstract interpretation.

We will prove that these two interpretations are related in a way that makes the abstract interpretation safe. The proof is based on logical relations which are relations that can be extended via a type structure and preserve fixpoints.

Let $\mathbf{D_s}$ and $\mathbf{D_a}$ be two domains. An admissible (or "inclusive" [Stoy 1977] and [Manna, Ness & Vuillemin 1972]) relation $\mathbf{R}$ between $\mathbf{D_s}$ and $\mathbf{D_a}$ is a subset $\mathbf{R} \subseteq \mathbf{D_s} \times \mathbf{D_a}$ such that for any directed subset $\mathbf{X} \subseteq \mathbf{R}$ the least upper bound $\bigsqcup \mathbf{X} \in \mathbf{R}$.

If an admissible relation $\mathbf{R} \subseteq \mathbf{D_s} \times \mathbf{D_a}$ relates bottom elements

$$\bot_\mathbf{s} \; \mathbf{R} \; \bot_\mathbf{a}$$

and two continuous functions $\mathbf{f_s} : \mathbf{D_s} \rightarrow \mathbf{D_s}$ and $\mathbf{f_a} : \mathbf{D_a} \rightarrow \mathbf{D_a}$ map related objects to related objects

$$\forall \, \mathbf{d_s} \in \mathbf{D_s}, \mathbf{d_a} \in \mathbf{D_a} : \mathbf{d_s} \; \mathbf{R} \; \mathbf{d_a} \Rightarrow \mathbf{f_s}(\mathbf{d_s}) \; \mathbf{R} \; \mathbf{f_a}(\mathbf{d_a})$$

then also the least fixpoints for these functions will be related

$$\mathbf{fix}(\mathbf{f_s}) \; \mathbf{R} \; \mathbf{fix}(\mathbf{f_a})$$

A relation between two domains can be extended to composite domains in a “logical” way [Reynolds 1983].

$$\mathbf{f_s} \; \mathbf{R_{X \rightarrow Y}} \; \mathbf{f_a} \Leftrightarrow \forall \, \mathbf{x_s}, \mathbf{x_a} : \mathbf{x_s} \; \mathbf{R_X} \; \mathbf{x_a} \Rightarrow \mathbf{f_s}(\mathbf{x_s}) \; \mathbf{R_Y} \; \mathbf{f_a}(\mathbf{x_a})$$
$$(\mathbf{x_s}, \mathbf{y_s}) \; \mathbf{R_{X \times Y}} \; (\mathbf{x_a}, \mathbf{y_a}) \Leftrightarrow \mathbf{x_s} \; \mathbf{R_X} \; \mathbf{x_a} \wedge \mathbf{y_s} \; \mathbf{R_Y} \; \mathbf{y_a}$$

Relations constructed in this way from admissible relations will also be admissible.

We will use the phrase *logical relations* to denote admissible relations which can be extended via a type structure.

3 A small language

The following little language will be used to illustrate the use of the attribute grammar framework for abstract interpretation. It is a flow-chart language with assignment statements and unrestricted jumps.

The language will be introduced with a context-free grammar and a discussion of its semantics. A formal semantics for the language will be given in the next section.

$$
\begin{array}{lcl}
\langle\textbf{program}\rangle & ::= & \langle\textbf{stmt}\rangle \\
\langle\textbf{stmt}\rangle & ::= & \langle\textbf{label}\rangle\ \textbf{:}\ \langle\textbf{stmt}\rangle \\
 & \mid & \textbf{begin}\ \langle\textbf{stmt}\rangle\ \textbf{;}\ \langle\textbf{stmt}\rangle\ \textbf{end} \\
 & \mid & \textbf{goto}\ \langle\textbf{label}\rangle \\
 & \mid & \langle\textbf{variable}\rangle\ \textbf{:=}\ \langle\textbf{exp}\rangle \\
 & \mid & \textbf{if}\ \langle\textbf{exp}\rangle\ \textbf{then}\ \langle\textbf{stmt}\rangle\ \textbf{else}\ \langle\textbf{stmt}\rangle \\
\langle\textbf{exp}\rangle & ::= & (\langle\textbf{exp}\rangle + \langle\textbf{exp}\rangle) \\
 & \mid & (\langle\textbf{exp}\rangle\ \textbf{-}\ \langle\textbf{exp}\rangle) \\
 & \mid & \langle\textbf{constant}\rangle \\
 & \mid & \langle\textbf{variable}\rangle
\end{array}
$$

The syntactic categories ⟨**variable**⟩ and ⟨**label**⟩ both denote identifiers (strings of letters and digits, starting with a letter) and ⟨**constants**⟩ denotes non negative numbers (a string of digits).

The semantics of the various language constructs are generally as expected. The **goto** command will of course normally send control to the command with the same label. If a label occurs more than once the **goto** command will select the first. A **goto** command to an undefined label causes a jump to the end of the program.

The output from the program is the contents of a given variable (say "x") at the end of the execution. This means that a program will end with an implicit **print**(x) command. Another possibility would have been to let the values of all variables be available at the end of the program but this would make a live-variable analysis less interesting.

4 Standard interpretation

This section introduces the standard interpretation for the language. In practice the design of the standard interpretation must go hand in hand with the abstract interpretation and the soundness proof. The general idea in the interpretations is to use a backward or continuation style semantics such that the meaning of a statement describes "the rest of the computation". In the case of the standard interpretation it is a mapping of the environment at the start of the statement to the final answer. For the abstract interpretation it is the set of variables which may be used in this or later statements along possible execution paths. A variable is said to be *live* at a given program point if it may be used in any execution path before it is assigned a new value. The relationship between these two interpretations can informally be stated as: if a variable is not live (*ie.* dead) then the standard interpretation will not change if that variable is made undefined. We will formalise this in section 6.

4.1 Attributed scheme

The attribute scheme will show the flow of information in the program but will leave the actual interpretation unspecified. A number of operator symbols will at a later stage be given a meaning which will define either the standard or the abstract interpretation.

The scheme defines a continuation style (or backward) semantics (see [Gordon 1979]). The attributes for statements (⟨**stmt**⟩) has two parts where the second part is used to send information from a labelled statement to a **goto** with that label. The first part is the actual meaning and the attributes is linked in such a way that the synthesised attribute is connected to the inherited attribute of its predecessor.

Notice that the attribute dependency is circular, as the attribute rule for ⟨**stmt**⟩ in the first production says the inherited attribute is defined from the synthesised one.

$\langle\textbf{program}\rangle$	$\{\$1.2\}$
$\quad = \langle\textbf{stmt}\rangle$	$\{\textbf{Final}, \$1.2\}$
$\langle\textbf{stmt}\rangle$	$\{\$3.1, \$3.2[\$1 \mapsto \$3.1]\}$
$\quad = \langle\textbf{name}\rangle\ \textbf{:}\ \langle\textbf{stmt}\rangle$	$\{\$0.1, \$0.2\}$
$\langle\textbf{stmt}\rangle$	$\{\$2.1, \textbf{Join}(\$2.2, \$4.2)\}$
$\quad = \textbf{begin}\ \langle\textbf{stmt}\rangle$	$\{\$4.1, \$0.2\}$
$\qquad \textbf{;}\ \langle\textbf{stmt}\rangle$	$\{\$0.1, \$0.2\}$
$\qquad \textbf{end}$	
$\langle\textbf{stmt}\rangle$	$\{\textbf{Goto}(\$0.2, \$2), \textbf{Nil}\}$
$\quad = \textbf{goto}\ \langle\textbf{name}\rangle$	

$\langle\textbf{stmt}\rangle$ $\{\textbf{Update}(\$0.1, \$1, \$3), \textbf{Nil}\}$
  $= \langle\textbf{name}\rangle \textbf{ := } \langle\textbf{exp}\rangle$

$\langle\textbf{stmt}\rangle$ $\{\textbf{If}(\$2, \$4.1, \$6.1), \textbf{Join}(\$4.2, \$6.2)\}$
  $= \textbf{if } \langle\textbf{exp}\rangle \textbf{ then } \langle\textbf{stmt}\rangle$ $\{\$0.1, \$0.2\}$
    $\textbf{else } \langle\textbf{stmt}\rangle$ $\{\$0.1, \$0.2\}$

$\langle\textbf{exp}\rangle$ $\{\textbf{Add}(\$2, \$4)\}$
  $= \textbf{(} \langle\textbf{exp}\rangle + \langle\textbf{exp}\rangle \textbf{)}$

$\langle\textbf{exp}\rangle$ $\{\textbf{Sub}(\$2, \$4)\}$
  $= \textbf{(} \langle\textbf{exp}\rangle \textbf{ - } \langle\textbf{exp}\rangle \textbf{)}$

$\langle\textbf{exp}\rangle$ $\{\textbf{Const}(\$1)\}$
  $= \langle\textbf{number}\rangle$

$\langle\textbf{exp}\rangle$ $\{\textbf{Var}(\$1)\}$
  $= \langle\textbf{name}\rangle$

The actual flow of information between labelled commands and **goto**'s is not described in the above scheme. This flow can be specified independent of the interpretations with these definitions

$$\textbf{Goto}(\mathbf{E}, \mathbf{l}) = \textbf{if } \mathbf{E}(\mathbf{l}) = \textbf{Undef then Final else } \mathbf{E}(\mathbf{l})$$
$$\textbf{Nil} = \lambda\, \mathbf{l}.\textbf{Undef}$$
$$\textbf{Join}(\mathbf{E}_1, \mathbf{E}_2) = \lambda\, \mathbf{l}.\ \textbf{if } \mathbf{E}_1(\mathbf{l}) = \textbf{Undef then } \mathbf{E}_2(\mathbf{l}) \textbf{ else } \mathbf{E}_1(\mathbf{l})$$

where **Undef** is a special symbol to indicate that no labels have been defined.

It is typical for this type of specification that the nitty-gritty details of actions with undefined labels must be made explicit. They can not be left undefined or to convention.

The attributed scheme can be seen as a polymorphic function definition. It is possible to type-check the scheme and give types to the operator symbols. Let $\mathbf{M}, \mathbf{D}, \mathbf{U}, \mathbf{A}, \mathbf{N}$ be type variables. The attributes will then have the following types:

synthesised of $\langle\textbf{stmt}\rangle : \mathbf{M} \times (\mathbf{A} \to (\mathbf{M} + \mathbf{U}))$
inherited of $\langle\textbf{stmt}\rangle : \mathbf{M} \times (\mathbf{A} \to (\mathbf{M} + \mathbf{U}))$
synthesised of $\langle\textbf{exp}\rangle : \mathbf{D}$

The types of the defined symbols are

$$\begin{array}{ll}
\mathbf{Goto} & : (\mathbf{A} \rightarrow (\mathbf{M} + \mathbf{U})) \times \mathbf{A} \rightarrow \mathbf{M} \\
\mathbf{Nil} & : \mathbf{A} \rightarrow (\mathbf{M} + \mathbf{U}) \\
\mathbf{Join} & : (\mathbf{A} \rightarrow (\mathbf{M} + \mathbf{U})) \times (\mathbf{A} \rightarrow (\mathbf{M} + \mathbf{U})) \rightarrow (\mathbf{A} \rightarrow (\mathbf{M} + \mathbf{U})) \\
\mathbf{Undef} & : \mathbf{U}
\end{array}$$

and the types for the operator symbols are

$$\begin{array}{ll}
\mathbf{Update} & : \mathbf{M} \times \mathbf{A} \times \mathbf{D} \rightarrow \mathbf{M} \\
\mathbf{If} & : \mathbf{D} \times \mathbf{M} \times \mathbf{M} \rightarrow \mathbf{M} \\
\mathbf{Final} & : \mathbf{M} \\
\mathbf{Add} & : \mathbf{D} \times \mathbf{D} \rightarrow \mathbf{D} \\
\mathbf{Sub} & : \mathbf{D} \times \mathbf{D} \rightarrow \mathbf{D} \\
\mathbf{Const} & : \mathbf{N} \rightarrow \mathbf{D} \\
\mathbf{Var} & : \mathbf{A} \rightarrow \mathbf{D}
\end{array}$$

In the interpretations the type variables **M** and **D** can vary their meaning. **U** will always be the singleton set $\{\mathbf{Undef}\}$, **N** is the domain of integers $\mathbb{Z}$, and **A** the domain of identifiers $\mathbb{A}$.

In both interpretations **M** and **D** will be given the same interpretation. Their intuitive meaning is different—respectively the meaning of "the rest of the program" and the meaning of "the expression"—so we will keep them separate in the interpretations and the proof. It is possible to define an abstract interpretation where these types are bound to different domains.

4.2 Standard interpretation

In the standard interpretation the meaning (of type **M**) of a statement will be a mapping of the environment before the statement to the final result. The final result is the value of the variable "**x**" at the end of the program.

For expressions the meaning (of type **D**) is a mapping of the environment to the value of the expression.

The standard interpretation uses the following interpretation of type names:

$$\begin{array}{l}
\mathbf{M} : \mathbf{env} \rightarrow \mathbb{Z} \\
\mathbf{D} : \mathbf{env} \rightarrow \mathbb{Z} \\
\mathbf{env} : \mathbb{A} \rightarrow \mathbb{Z}
\end{array}$$

and the interpretation of operator symbols are

$$
\begin{aligned}
&\mathbf{Update}(\mathbf{Cn},\mathbf{n},\mathbf{e}) &&= \lambda\,\mathbf{x}.\mathbf{Cn}(\mathbf{x}[\mathbf{n}\mapsto\mathbf{e}(\mathbf{x})])\\
&\mathbf{If}(\mathbf{e},\mathbf{Ct},\mathbf{Cf}) &&= \lambda\,\mathbf{x}.\ \mathbf{if}\ \mathbf{e}(\mathbf{x})\ \mathbf{then}\ \mathbf{Ct}(\mathbf{e})\ \mathbf{else}\ \mathbf{Cf}(\mathbf{e})\\
&\mathbf{Final} &&= \lambda\,\mathbf{e}.\mathbf{e}(\text{“x”})\\
&\mathbf{Add}(\mathbf{e}1,\mathbf{e}2) &&= \lambda\,\mathbf{e}.\mathbf{e}1(\mathbf{e})+\mathbf{e}2(\mathbf{e})\\
&\mathbf{Sub}(\mathbf{e}1,\mathbf{e}2) &&= \lambda\,\mathbf{e}.\mathbf{e}1(\mathbf{e})-\mathbf{e}2(\mathbf{e})\\
&\mathbf{Const}(\mathbf{n}) &&= \lambda\,\mathbf{e}.\mathbf{n}\\
&\mathbf{Var}(\mathbf{v}) &&= \lambda\,\mathbf{e}.\mathbf{e}(\mathbf{v})
\end{aligned}
$$

5 Live-variable analysis

The live-variable analysis will be specified as an interpretation of the above attributed scheme. The meaning of a statement is the set of live variables at the start of the statement, and the meaning of an expression is the set of variables used in it.

The interpretation of type names is

$$
\begin{aligned}
&\mathbf{M} : \mathbb{P}(\mathbb{A})\\
&\mathbf{D} : \mathbb{P}(\mathbb{A})
\end{aligned}
$$

This is the powerset of variable names ordered by set inclusion. The interpretation of operator symbols is

$$
\begin{aligned}
&\mathbf{Update}(\mathbf{Cn},\mathbf{n},\mathbf{e}) &&= \mathbf{e}\cup(\mathbf{Cn}\backslash\{\mathbf{n}\})\\
&\mathbf{If}(\mathbf{e},\mathbf{Ct},\mathbf{Cf}) &&= \mathbf{e}\cup\mathbf{Ct}\cup\mathbf{Cf}\\
&\mathbf{Final} &&= \{\text{“x”}\}\\
&\mathbf{Add}(\mathbf{e}1,\mathbf{e}2) &&= \mathbf{e}1\cup\mathbf{e}2\\
&\mathbf{Sub}(\mathbf{e}1,\mathbf{e}2) &&= \mathbf{e}1\cup\mathbf{e}2\\
&\mathbf{Const}(\mathbf{n}) &&= \varnothing\\
&\mathbf{Var}(\mathbf{v}) &&= \{\mathbf{v}\}
\end{aligned}
$$

All these functions can easily be seen to be continuous. It is not difficult to define attribute functions that are not continuous as the set difference operation ($\backslash$) is not continuous. In this case it only subtracts a singleton set and that operation is continuous.

6 Correctness proof of the abstract interpretation

In the correctness proof we will introduce a relationship between values in the two interpretations of the base types. The index **s** will be used for the standard interpretation and **a** for the abstract interpretation. Thus we will establish a relation $\trianglerighteq_{\mathbf{M}}$ between $\mathbf{M_s}$ and $\mathbf{M_a}$ and a relation $\trianglerighteq_{\mathbf{D}}$ between $\mathbf{D_s}$ and $\mathbf{D_a}$. The relation can be extended to other types in our type structure as described in section 2.

The proof is in two parts. In the first part we prove that the two interpretations of the operator symbols can be related. In the second part we prove that the relation is admissible and hence can be extended to fixpoints. From this we conclude that attributes in the two interpretations will be related for all programs.

To prove a relation between two interpretations of the language we must make sure that

- all attribute rules are continuous functions.
- a relation holds between values of the base types (**M** and **D**).
- that the relations relate the interpretations of the operator symbols
- that the relation is *logical* and hence can be extended to composite types and to fixpoints.

6.1 Relation

The relation for **M** and **D** is the same and is defined as

$$\mathbf{c_s} \trianglerighteq_{\mathbf{M}} \mathbf{c_a} \Leftrightarrow \forall\, \mathbf{x}.\mathbf{x} \notin \mathbf{c_a} \Rightarrow \forall\, \mathbf{e}.\mathbf{c_s}(\mathbf{e}) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \perp])$$

This can be interpreted as: if a variable is not live then the standard interpretation of the rest of the program will not change if its value is undefined. The interpretation for **D** is the same except "the rest of the program" should be "the whole of the expression".

The relation does not say that a live variable will be used in later statements but only that if a variable is not live, it will not be used.

6.2 Local correctness

The correctness proof requires that the relation is proved for the interpretations of the operator symbol. The proof is essentially no more than symbol manipulation. It should be done for all the operator symbols but we will only include the proof for the **Update**.

We will prove that

$$\mathbf{Update_s} \trianglerighteq_{(\mathbf{M}\times\mathbf{A}\times\mathbf{D})\to\mathbf{M}} \mathbf{Update_a}$$

or equivalently that

$$\begin{aligned}&\forall\, \mathbf{c_s}, \mathbf{c_a}, \mathbf{v_s}, \mathbf{v_a}, \mathbf{e_s}, \mathbf{e_a} :\\ &\quad \mathbf{c_s} \trianglerighteq_{\mathbf{M}} \mathbf{c_a}, \mathbf{v_s} = \mathbf{v_a}, \mathbf{e_s} \trianglerighteq \mathbf{e_a} \Rightarrow \mathbf{Update_s}(\mathbf{c_s}, \mathbf{v_s}, \mathbf{e_s}) \trianglerighteq_{\mathbf{M}} \mathbf{Update_a}(\mathbf{c_a}, \mathbf{v_a}, \mathbf{e_a})\end{aligned}$$

Let $\mathbf{c_s} \in \mathbf{M_s}, \mathbf{c_a} \in \mathbf{M_a}, \mathbf{v_s} \in \mathbb{A}, \mathbf{v_a} \in \mathbb{A}, \mathbf{e_s} \in \mathbf{D_s}, \mathbf{e_a} \in \mathbf{D_a}$ be any values such that

$$\mathbf{c_s} \trianglerighteq_{\mathbf{M}} \mathbf{c_a}, \mathbf{v_s} = \mathbf{v_a}, \mathbf{e_s} \trianglerighteq \mathbf{e_a}$$

We need to prove that

$$\mathbf{Update_s}(\mathbf{c_s}, \mathbf{v_s}, \mathbf{e_s}) \trianglerighteq_{\mathbf{M}} \mathbf{Update_a}(\mathbf{c_a}, \mathbf{v_a}, \mathbf{e_a})$$

Let $\mathbf{v} = \mathbf{v_a} = \mathbf{v_s}$ and let $\mathbf{x}$ be any variable such that $\mathbf{x} \notin \mathbf{e_a}$ and $\mathbf{e}$ any environment $\mathbf{e} \in \mathbf{env_s}$. The relationship we seek is then

$$\mathbf{c_s}(\mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})]) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \bot][\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e}[\mathbf{x} \mapsto \bot])])$$

using $\mathbf{e_s}(\mathbf{e}) = \mathbf{e_s}(\mathbf{e}[\mathbf{x} \mapsto \bot])$ gives

$$\mathbf{c_s}(\mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})]) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \bot][\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})])$$

if $\mathbf{v} = \mathbf{x}$ then there is no more to prove; else assume $\mathbf{x} \notin \mathbf{c_a}$

$$\mathbf{c_s}(\mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})]) = \mathbf{c_s}(\mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})][\mathbf{x} \mapsto \bot])$$

and with $\mathbf{e}_1 = \mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})]$ we have

$$\mathbf{c_s}(\mathbf{e}_1) = \mathbf{c_s}(\mathbf{e}_1[\mathbf{x} \mapsto \bot])$$

This is true by the assumption $(\mathbf{c_s} \trianglerighteq_{\mathbf{M}} \mathbf{c_a} \wedge \mathbf{x} \notin \mathbf{c_a})$.

The proof for the other operator symbols is performed in the same fashion.

6.3 Fixpoint induction

The final part of the proof is to show that the relation $\trianglerighteq_{\mathbf{M}}$ is admissible. That is to prove that for any directed subset $\mathbf{X} \subseteq \trianglerighteq_{\mathbf{M}}$ that $\bigsqcup \mathbf{X} \subseteq \trianglerighteq_{\mathbf{M}}$. This follows from the fact that $\mathbf{M_a}$ is finite (there is only a finite number of variables in a program) and that $\forall\, \mathbf{e}.\mathbf{c_s}(\mathbf{e}) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \bot])$ is an admissible predicate on $\mathbf{M_s}$. This follows from simple structural argument due to [Manna, Ness & Vuillemin 1972] (see also [Stoy 1977].

7 Implementation of the attribute grammars

The attribute scheme presented in section 4 is clearly circular but both the standard semantics and the abstract interpretation can easily be implemented. The circularity in the attributed scheme occurs for the second part of the synthesised attribute to $\langle\mathbf{stmt}\rangle$. This part has type $\mathbf{A} \rightarrow (\mathbf{M} + \mathbf{U})$. The fixpoint iteration will be performed in different ways in the two interpretations.

In the standard interpretation the recursively defined attribute has type $\mathbb{A} \rightarrow (\mathbf{env} \rightarrow \mathbb{Z})$. This is a functional object and it is therefore just a definition of a recursive function.

For the abstract interpretation the circular definition is for objects of type $\mathbb{A} \rightarrow \mathbb{P}(\mathbb{A})$. The evaluation of this type of object will require fixpoint iteration but it is guaranteed to terminate as a program can only contain a finite number of labels and a finite number of variables. The objects can be represented as a list of sets—one for each variable in the program and iteration continues until no more variable names can be included in the sets. Other and more efficient implementations are also possible but this will not be discussed further here.

We have implemented these two interpretation using a prototype system that takes an attributed scheme together with definitions of the operator symbols and produces an attribute evaluater where attributes are evaluated in lazy higher order functional language and where fixpoint iteration algorithms are used for non-flat finite domains.

8 Conclusion

The use of attribute grammars for the specification of abstract interpretation can be seen as an alternative notation to denotational semantics. Some of the advantages with this framework are:

- An attribute grammar can be based directly on the concrete syntax removing the need for abstract syntax and intermediate levels of parsing.
- A change from forward to backward-style semantics makes a big difference in denotational semantics but only means a simple rewiring of an attributed scheme. It is in this framework possible to specify forward and backward analyses as well as combinations hereof.
- As the same framework can be used for code generation it should be possible to use the information obtained from an abstract interpretation to make an optimising compiler.
- The implementation can be made directly from the specification.

Acknowledgements

Thanks to Alan Mycroft and the referees for comments on drafts of this paper.

References

[Abramsky & Hankin 1987] S Abramsky and C Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.

[Aho, Sethi & Ullman 1986] A V Aho, R Sethi and J D Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Babich & Jazayeri 1978] W A Babich and M Jazayeri. *The Method of Attributes for Data Flow Analysis (part 1 and 2).* Acta Inf. **10**, pp. 245–272, 1978.

[Chirica & Martin 1979] L M Chirica and D F Martin. *An Order-Algebraic Definition of Knuthian Semantics.* Math. Systems Theory **13**, pp. 1–27, 1979.

[Courcelle & Deransart 1988] B Courcelle and P Deransart. *Proofs of partial correctness for attribute grammars with applications to recursive procedures and logic programming.* Inform. and Comput. **78**(1), pp. 1–55, July, 1988.

[Cousot & Cousot 1977] P Cousot and R Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In *4th POPL, Los Angeles, CA*, pp. 238–252, Jan., 1977.

[Deransart 1983] P Deransart. *Logical Attribute Grammars.* In *IFIP'83, Paris, France*, pp. 463–469. North-Holland, Sept., 1983.

[Deransart, Jourdan & Lorho 1988] P Deransart, M Jourdan and B Lorho. *Attribute grammars. Definitions, systems, and bibliography.* Volume 323 of LNCS. Springer-Verlag, 1988.

[Farrow 1986] R Farrow. *Automatic Generation of Fixed-Point-Finding Evaluators for Circular but Well-Defined Attribute Grammars.* In *SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, California*, pp. 85–98. Volume 21(7) of ACM SIGPLAN Not., July, 1986.

[Gallier 1984] J Gallier. *An Efficient Evaluator for Attribute Grammars with Conditionals.* Tech. Rep. MS-CIS-83-36. Philadelphia, PA, May, 1984.

[Gordon 1979] M J C Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, 1979.

[Johnson 1975] S C Johnson. *YACC: Yet Another Compiler-Compiler.* Manual. Murray Hill, NJ, 1975.

[Jones & Simon 1986] L G Jones and J Simon. *Hierarchical VLSI Design Systems Based on Attribute Grammars.* In *13th POPL, St. Petersburg, Florida*, pp. 58–69, Jan., 1986.

[Knuth 1968] D E Knuth. *Semantics of Context-Free Languages.* Math. Systems Theory **2**(2), pp. 127–145, June, 1968. Correction ibid 5(1):95–96 Mar. 1971.

[Knuth 1971] D E Knuth. *Examples of formal semantics.* In *Symposium on semantics of algorithmic languages* (E Engeler, ed.), pp. 212–235. Volume 188 of Lect. Notes in Math. Springer-Verlag, 1971.

[Manna, Ness & Vuillemin 1972] Z Manna, S Ness and J Vuillemin. *Inductive methods for proving properties of programs.* In *ACM Conference on Proving Assertions About Programs*, pp. 27–50. Volume 7(1) of ACM SIGPLAN Not., Jan., 1972.

[Mayoh 1981] B H Mayoh. *Attribute Grammars and Mathematical Semantics.* SIAM J. Comp. **10**(3), pp. 503–518, Aug., 1981.

[Morel 1984] E Morel. *Data Flow Analysis and Global Optimization.* In *Methods and Tools for Compiler Construction* (B Lorho, ed.), pp. 289–315. Cambridge Univ. Press, 1984.

[Mycroft & Jones 1986] A Mycroft and N D Jones. *A Relational Framework for Abstract Interpretation.* In *Programs as Data Objects* (H Ganzinger and N D Jones, eds.), pp. 156–171. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.

[Nielson & Nielson 1988] F Nielson and H R Nielson. *Two-level semantics and code generation.* Theor. Comp. Sci. **56**(1), pp. 59–133, Jan., 1988.

[Paulson 1982] L Paulson. *A Semantics-Directed Compiler Generator.* In *9th POPL, Albuquerque, NM*, pp. 224–233, Jan., 1982.

[Paulson 1984] L Paulson. *Compiler Generation from Denotational Semantics.* In *Methods and Tools for Compiler Construction* (B Lorho, ed.), pp. 219–250. Cambridge Univ. Press, 1984.

[Reynolds 1983] J C Reynolds. *Types, abstraction and parametric polymorphism.* In *IFIP'83, Paris, France.* North-Holland, Sept., 1983.

[Schmidt 1986] D A Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Newton, MA, 1986.

[Stoy 1977] J E Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Watt & Madsen 1983] D A Watt and O L Madsen. *Extended Attribute Grammars.* Comput. J. **26**(2), pp. 142–153, 1983.

[Wilhelm 1981] R Wilhelm. *Global Flow Analysis and Optimization in the MUG2 Compiler Generating System.* In *Program Flow Analysis: Theory and Applications* (S S Muchnick and N D Jones, eds.), chapter 5, pp. 132–159. Prentice-Hall, 1981.