A framework for construction and evaluation of high-level specifications for program analysis techniques

G A Venkatesh
University of Wisconsin - Madison
venky@cs.wisc.edu

Abstract

Abstract interpretation introduced the notion of formal specification of program analyses. Denotational frameworks are convenient for reasoning about such specifications. However, implementation considerations make denotational specifications complex and hard to develop. We present a framework that facilitates the construction and understanding of denotational specifications for program analysis techniques. The framework is exemplified by specifications for program analysis techniques from the literature and from our own research. This approach allows program analysis techniques to be incorporated into automatically generated program synthesizers by including their specifications with the language definition.

1. Introduction

Recent developments in programming environments have blurred the distinction between language editors and compilers/interpreters. Modern program synthesizers carry out many tasks that were once traditionally considered as compiler functions. These tasks include syntax checking, type checking, type inference, and various data flow analyses. The interactive nature of these synthesizers exploits these techniques to provide an effective program synthesis environment. The growing sophistication of these analysis techniques necessitates a structured approach to their design to ease their development as well as to ensure their correctness.

Most flow analysis techniques are currently developed as algorithms in which implementation details are combined with the technique itself. This has several disadvantages. First, the design process is complicated by the need to handle low-level implementation details. The design of flow analysis techniques is heavily dependent on the application. Generally, flow analysis techniques provide approximate information about the dynamic behavior of programs. There is always a trade-off between the effort involved in the analysis and the precision with which information is obtained from the analysis. The application sets the requirements for precision and efficiency. Low-level development of an analysis technique makes it difficult to prevent duplication of effort in developing techniques that are closely related.

Second, the presence of implementation details makes it harder to understand the algorithms in order to maintain or improve existing ones. Last, correctness proofs are difficult to derive for algorithms with too much detail. To gain confidence about an analysis, it is necessary to establish certain desirable properties of the analysis. Three essential properties are:

(1) Consistency with the semantics of the language:

Program analysis techniques are designed to make assertions about properties (static and/or dynamic) of the program. Since such properties are determined by the semantics of the language, it is important to ensure that the analysis techniques are consistent with the language semantics. This becomes crucial as the analyses become more sophisticated.

(2) Termination:

Static analysis algorithms must terminate for any program. It is useful to provide formal reasoning that guarantees termination.

This work was supported by National Science Foundation under grant CCR 87-06329.

Author's address: Computer Sciences Department, University of Wisconsin, 1210 W.Dayton, Madison, WI 53706

(3) Safety:

A compile-time analysis usually provides an approximation to the actual execution of the program either due to inherent limits such as unknown input values or due to implementation considerations that trade-off precision for efficiency and/or termination. The analysis is designed to err on the conservative side. The actual runtime property of the program (that is being approximated) must imply the information gathered from the analysis.

1.1. High-level specifications

The seminal paper on *abstract interpretation* [2] introduced the notion that a wide variety of program analysis techniques could be specified formally as interpretations over abstract domains. These formal specifications can then be used to derive the required correctness proofs. Although the original work involved operational specifications for flow-chart languages, most of the later studies based on abstract interpretation [10, 8, 6, 4, etc.] have used denotational specifications.

A denotational specification of a program analysis can be considered as an alternate semantics for the language. Most of the techniques developed for providing standard semantics of a language can be used to provide an alternate semantics. The structural and compositional nature of denotational specifications ease the development of correctness proofs through the use of well understood methods such as structural induction, fixed-point induction, etc. Consistency with standard semantics can be established easily with respect to the denotational semantics of the language. Studies such as [11, 12, 1, 15] demonstrate the possibility of providing efficient evaluators for denotational specifications.

1.2. Problems with denotational frameworks

The use of denotational frameworks for specification of program analysis techniques tends to be less than ideal since the denotational functions make assertions only about the relationship between the input and output values. However, program analysis often requires information at various program points that correspond to results from partial evaluations of the denotational function representing the meaning of the program. To make this possible one must explicitly introduce parameters to these functions to *cache* the intermediate values [10, 5]. This results in messy specifications that are difficult to understand. It also requires a wasteful duplication of effort in maintaining the cache parameter. This problem is analogous to the use of copy attributes in attribute grammar frameworks to propagate information between nodes in the abstract syntax tree.

Moreover, the use of caching makes definite assumptions about the evaluation schemes for the specifications. It is no longer sufficient to show the correctness of the evaluator through its consistency with the relationship between inputs and outputs. For the information obtained from an evaluation to be meaningful, the evaluator must also be consistent with the assumptions made about the intermediate states possible in the evaluations. It is necessary to formalize such assumptions to prove the correctness of the analysis.

1.3. Solutions

This paper describes a framework that facilitates the development of program analysis techniques through denotational specifications. Features that are common to most analysis techniques are incorporated into the framework to avoid duplication of effort. For example, a facility is provided to specify collection of results of partial evaluations in an implicit fashion. The operators used in the specification language are defined formally in an axiomatic framework. This allows formal reasoning about results from partial evaluations without over-specification of possible evaluation schemes for the denotational specifications.

Our approach allows multiple analysis specifications for any language and a single interpretation model. All application dependent techniques used in a particular analysis for efficiency considerations (including termination) must be included in the specification for that analysis. The interpretation model is independent of the analysis as well as the language in which the programs to be analyzed are written. An implementation for the model may include efficient evaluation techniques such as incremental evaluation. The correctness of the interpretation model is proved independent of any particular specification.

2. The framework

The program analysis specifications are written in a fashion similar to denotational specification of standard semantics. A specification consists of domain declarations and semantic functions over these domains. Several features are included in the specification language to support the techniques required in most flow analysis designs. This section provides a brief description of such features. For clarity, we will use symbols that are traditionally used in denotational semantics although the concrete syntax for the specifications would have direct translations for those symbols for machine readability.

2.1. Domain declarations

Domains

primitive domain ::=
integer | boolean | ordinal | syntactic | label

domain ::= *primitive domain*
| *domain* × *domain*
| *domain* + *domain*
| **lifted** *domain*
| **topped** *domain*
| **powerset of** *domain*
| **[recursive] store** *domain* → *domain*
| **function** *function type*

function type ::=
[[strict] *domain*[*] { → **[strict]** *domain* }]
→ *domain*
[collect [powerset of] *domain* [↓ n]]
[merge cache]

The *lifted* domain introduces a special element ⊥ as the least element into the base domain. The *topped* domain introduces a special element ⊤ as the greatest element into the base domain. Lookup and update functions are predefined for a *store* domain. The lookup is denoted by s[x] while the update is denoted by $s[y_1/x_1, ..., y_n/x_n]$. An update for all elements from a set is denoted by **s[foreach x in S** f(x)/x]. In a *recursive store*, the store can be stored as a value in itself. $\perp_s$ denotes the least element of a *lifted store* domain s. $\perp_s[\,]$ denotes an empty store from the domain s.

A function can be declared as *strict* in any of its arguments. Assuming that the least elements exist in the corresponding domains (including the result domain), the framework will internally add a prefix to the function definition to make it strict in those arguments.

A major goal of the framework is to avoid the explicit caching of intermediate values ("states") in the semantic function definitions. The **collect** option for a semantic function associates with the function an extra argument and an additional component to the result. Both of them have the type **store** *label* → *domain*. All syntactic objects are automatically provided with labels by the framework. Depending on whether the flow analysis requires an input state or an output state to be associated with the syntactic object, the corresponding domain is specified in the **collect** option. Whenever a semantic function is evaluated, the value of the specified argument (or result) is joined with the previous value in the cache using the join operator (∇) for that domain. The **powerset of** option changes the type of cache to **store** *label* → **powerset of** *domain*. The intermediate values are converted into sets before the join (∇) operator in the powerset domain is applied.

For a function for which the **collect** option is not specified, the user may define how the output cache is to be obtained if any of the arguments to the function have implicit caches associated with them. However, in many cases the output cache is merely a join of all the input caches and a possible cache from the evaluation of the function. The **merge cache** option specifies that this is to be the default. The cache associated with argument i can be explicitly accessed as cache$i. In a semantic function definition $.cache refers to the implicit cache argument.

All the primitive domains except *ordinal* are flat domains. *ordinal* has a total ordering induced by the arithmetic ≤ relation. The standard operations induce orderings on the compound domains. However, a user may explicitly define an ordering to be used for a domain. The framework will not check whether the ordering definition produces a partial ordering. It may be necessary to use equivalence classes with respect to the defined ordering. The *representative* declaration specifies the representative for the equivalence class to be used in the implementation.

Ordering

primitive ordering ::=
subset | flat | arithmetic

ordering ::= *primitive ordering*
| *logical expression*

Representative

representative ::= convex-closure
| *expression*
| any

2.2. Function definitions

Functions are defined as expressions in the language described below. The cache option in the fixed-point expression denotes that the functional is monotonic in the implicit cache argument (but not necessarily in other arguments). The user must ensure that the least fixed point is finitely computable.

---

expression ::= *primitive functions*
| *id* | *constant*
| *store function*
| λ*id.expression*
| *expression* ↓ *constant*
| *expression expression*
| *(expression , expression)*
| *expression* ∘ *expression*
| [*expression*] ∇ *expression*
| *expression* → *expression , expression*
| **fix [cache]** expression
| **let** *id* = *expression*
{**and** *id* = *expression*} **in** *expression*

2.3. Interpretation model

The semantics of the specification language is provided through a formalism based on axiomatic rules. Any evaluation scheme used to provide an interpreter for the specifications must be consistent with these definitions. These rules also provide a formal definition for the collection of results of partial evaluations and can be used to reason about the correctness of the intermediate states. The choice of an axiomatic system allows such a formal definition while avoiding over-specification of possible evaluation schemes. A sample of such definitions are provided below. The complete specification of the interpretation model can be found in [16].

(1) defines the evaluation of a semantic function for which the collect option has been specified. The cache parameter is invisible in the specifications but can be implemented as an additional parameter to the function that is maintained automatically. (2) defines the composition operator for a function that does not require collection of intermediate results. However, it defines the existence of the intermediate value t. (3) defines the join (∇) operator for functions when the collect option is used. F denotes the environment of function bindings under which the evaluation is carried out. T denotes the predicates for type checking that is carried out statically.

---

(1) $$\frac{T, F \vdash fx \Rightarrow y,\ \vdash c' = c[(x \times y)\downarrow n/f.label] \nabla c}{F \vdash fx\langle c\rangle \Rightarrow y\langle c'\rangle}$$

(2) $$\frac{T, F \vdash f_2\, x \Rightarrow t,\ F \vdash f_1\, t \Rightarrow y}{F \vdash f_1 \circ f_2\, x \Rightarrow y}$$

(3) $$\frac{T, F \vdash f_1\, x\, \langle c\rangle \Rightarrow t_1\, \langle c_1\rangle,\ F \vdash f_2\, x\, \langle c\rangle \Rightarrow t_2\, \langle c_2\rangle,\ \vdash y = t_1 \nabla t_2,\ \vdash c' = c_1 \nabla c_2}{F \vdash f_1 \nabla f_2\, x\, \langle c\rangle \Rightarrow y\, \langle c'\rangle}$$

An evaluation scheme for the model can use techniques such as incremental evaluation to provide an efficient interpreter for the specifications. Incremental evaluation of functions using caching has been proposed in [13]. However, use of aggregate values as inputs to functions can reduce the effectiveness of incremental evaluations. To allow for cases in which analysis dependent information can be used to provide better incremental evaluations, a feature is provided in the framework to mark instances of syntactic objects (and hence the semantic functions associated with them) as *must-be-evaluated*. If the analysis can dynamically mark the functions that need to be evaluated, then the functions, while unmarked, will be evaluated only once. The use of this feature is demonstrated in the first example specification.

3. Example specifications

The example specifications in this section have been selected to illustrate various features of the framework as well as to demonstrate the ease with which issues of precision and efficiency can be handled in the construction of program analysis specifications. The first example was motivated by the conditional constant propagation algorithms discussed by Wegman and Zadeck in [18]. The second example is for integer range analysis. The range analysis technique was motivated by research to extend type inferencing to automatically select type implementations for a given program. The correctness proofs for these specifications are beyond the scope of this paper. The formalism to express correctness of the range analysis specification can be found in [17]. The third example demonstrates the use of our framework in formulating the collecting interpretation model developed by Hudak and Young in [5] for an abstract functional language.

3.1. Abstract Syntax and Standard Semantics

For the first two examples we will use a small imperative language that will be sufficient to demonstrate the various issues. Fig. 1 provides the abstract syntax for the language. The language includes as base values integers and booleans. The identifiers can only be of integer types. For brevity, we will omit the syntactic domain of declarations. The standard semantics for the language is expressed in our framework in Fig. 2. In the following specifications, the conversions between base values in the language and their denotations are implicit.

```
    i ∈ Id                    Identifiers
    e ∈ Exp                   Expressions
    b ∈ B-Exp                 Boolean Expressions
    s ∈ Stat                  Statements
    p ∈ Prog                  Program

p ::= (Prog s)

s ::= (Assign i e)            Assignment statement
    | (If b s1 s2)            Conditional Statement
    | (While b s)             Loop statement
    | (Comp s1 s2)            Statement Composition

e ::= (Id id) | (Int_Lit n)
    | (Add e1 e2)

b ::= (Bool_Lit l)
    | (And b1 b2) | (Equal e1 e2)
```

Figure 1. Abstract Syntax

```
Domains:
Id, Exp, B-Exp, Stat, Prog = syntactic;
L_Integer = lifted integer;
L_Bool = lifted boolean;
Prog_Store = store [[Id]] → L_Integer;

Function Declarations:

E[[Exp]] :      Prog_Store → L_Integer;
B[[B-Exp]] :    Prog_Store → L_Bool;
S[[Stat]] :     Prog_Store → Prog_Store;
P[[Prog]] :     Prog_Store → Prog_Store;

Function Definitions:

P[[(Prog s)]] = λx. S[[s]]x;

S[[(Assign i e)]] = λx. x[(E[[e]]x) / [[i]]];
S[[(If b s1 s2)]] = λx. (B[[b]] → S[[s1]] , S[[s2]]) x;
S[[(While b s)]] = fix(λf. λx. (B[[b]] →
                        (f ∘ S[[s]]) , λy.y) x);
S[[(Comp s1 s2)]] = λx. (S[[s2]] ∘ S[[s1]]) x;

E[[(Id id)]] = λx.x[[[id]]];
E[[(Int_Lit n)]] = λx.L_Integer[[n]];
E[[(Add e1 e2)]] = λx.Add E[[e1]]x E[[e2]]x;

B[[(Bool_Lit l)]] = λx.L_Bool[[l]];
B[[(And b1 b2)]] = λx.And B[[b1]]x B[[b2]]x;
B[[(Equal e1 e2)]] = λx.Equal E[[e1]]x E[[e2]]x;
```

Figure 2. Standard Semantics

3.2. Conditional Constant Propagation

Conditional constant propagation algorithms can potentially discover more constants than the simple constant propagation algorithm developed by Kildall [9] by evaluating all conditional branches with all constant operands. Parts of the program that are never executed are ignored. Hence, assignments in those parts cannot kill potential constants. For example, consider the code segment below:

```
i := 1; if i = 1 then j := 1 else j := 2
```

Evaluation of the conditional can show that `j` is never assigned the value 2 and cannot affect the consideration of `j` as a potential constant.

We will first provide a specification (Fig. 3) that does a naive version of the conditional constant propagation. The specification corresponds to an analysis that essentially simulates the execution of the program in a simple domain.

`Cstore` provides a map between identifiers and values from the usual three-level lattice of constants, `Con`. $\perp$ in this lattice denotes that the identifier may or may not be a constant, while $\top$ denotes that the identifier is not a constant. The input to the

Domains:

```
Id, Exp, B-Exp, Stat, Prog = syntactic;
Con = topped lifted integer;
Cstore = lifted store [[Id]] --> Con;
```

Function Declarations:

```
E[[Exp]] :      Strict Cstore --> Con;
Bt[[B-Exp]] :   Strict Cstore --> Cstore;
Bf[[B-Exp]] :   Strict Cstore --> Cstore;
S[[Stat]] :     Strict Cstore --> Cstore;      (1)
P[[Prog]] :     --> Cstore;
```

Function Definitions:

$$\mathbf{P}[\![(\textbf{Prog } s)]\!] = \mathbf{S}[\![s]\!]\ \perp_{Cstore}[\,]; \qquad (2)$$

$$\mathbf{S}[\![(\textbf{Assign } i\ e)]\!] = \lambda x.\ \textbf{let } n = \mathbf{E}[\![e]\!]x \textbf{ in } x[(x[[\![i]\!]]\nabla n) / [\![i]\!]]; \qquad (3)$$

$$\mathbf{S}[\![(\textbf{If } b\ s_1\ s_2)]\!] = \lambda x.\ \textbf{let } bt = \mathbf{B}_t[\![b]\!]\ x \textbf{ and } bf = \mathbf{B}_f[\![b]\!]\ x \textbf{ in } \mathbf{S}[\![s_1]\!]\ bt\ \nabla\ \mathbf{S}[\![s_2]\!]\ bf;$$

$$\mathbf{S}[\![(\textbf{While } b\ s)]\!] = \textit{fix}(\lambda f.\ \lambda x.\ \textbf{let } bt = \mathbf{B}_t[\![b]\!]\ x \textbf{ and } bf = \mathbf{B}_f[\![b]\!]\ x \textbf{ in } ((f \circ \mathbf{S}[\![s]\!])\ bt)\ \nabla\ bf; \qquad (4)$$

$$\mathbf{S}[\![(\textbf{Comp } s_1\ s_2)]\!] = \lambda x.\ (\mathbf{S}[\![s_2]\!] \circ \mathbf{S}[\![s_1]\!])\ x;$$

$$\mathbf{E}[\![(\textbf{Id } id)]\!] = \lambda x.x[[\![id]\!]];$$

$$\mathbf{E}[\![(\textbf{Int\_Lit } n)]\!] = \lambda x.\mathrm{Con}[\![n]\!];$$

$$\mathbf{E}[\![(\textbf{Add } e_1\ e_2)]\!] = \lambda x.\ \textbf{let } n1 = \mathbf{E}[\![e_1]\!]\ x \textbf{ and } n2 = \mathbf{E}[\![e_2]\!]\ x \textbf{ in } (n1 = \top) \textbf{ or } (n2 = \top) \rightarrow \top\ ,\ \mathrm{Add}\ n1\ n2;$$

$$\mathbf{B}_t[\![(\textbf{Bool\_Lit } l)]\!] = \lambda x.\mathrm{boolean}[\![l]\!] \rightarrow x\ ,\ \perp;$$

$$\mathbf{B}_t[\![(\textbf{And } b_1\ b_2)]\!] = \lambda x.\ \textbf{let } bt1 = \mathbf{B}_t[\![b_1]\!]\ x \textbf{ and } bt2 = \mathbf{B}_t[\![b_2]\!]\ x \textbf{ in } (bt1 = \perp) \textbf{ or } (bt2 = \perp) \rightarrow \perp\ ,\ x;$$

$$\mathbf{B}_t[\![(\textbf{Equal } e_1\ e_2)]\!] = \lambda x.\ \textbf{let } n1 = \mathbf{E}[\![e_1]\!]\ x \textbf{ and } n2 = \mathbf{E}[\![e_2]\!]\ x \textbf{ in } (n1 = \top) \textbf{ or } (n2 = \top) \textbf{ or } (n1 = n2) \rightarrow x\ ,\ \perp;$$

Figure 3. Conditional Constant Propagation (Naive)

program is an empty store. Identifiers mapped to $\top$ in the output are not constants. An identifier mapped to an integer is a constant with that value. Identifiers are not mapped to $\perp$ unless they were used before their definition in the program.

The least element ($\perp$) of `Cstore` denotes an unreachable state. As all the semantic functions are declared to be strict in their arguments, information from program parts that are unreachable are never propagated. Unreachable states arise from the evaluation of the semantic functions for boolean expressions. The semantic function $\mathbf{B}_t$ outputs $\perp$ if the boolean expression contains all constant operands and evaluates to *false*. For all other expressions, it is an identity function. The semantic function $\mathbf{B}_f$ is similar but outputs $\perp$ when the expression evaluates to the constant *true*. The specification of $\mathbf{B}_f$ is very similar to that of $\mathbf{B}_t$ and has been left out.

3.2.1. Improving Precision

As explained in §3.2, although this analysis can potentially detect more constants than Kildall's, it does not use information from conditions that contain identifiers that are constants locally but not globally. For example consider the code segment below:

```
i := 1; i := 2; if i=2 then j := 1 else j := 2;
```

The analysis does not use the fact that `i` is locally a constant in the conditional and assumes that both the branches are executed. To improve this analysis to handle local constants, only the four numbered lines in Fig.3. need to be modified. The modifications are shown in Fig.4.

The semantic function **S** is modified to output the current assignments to the identifiers rather than a join of all previous assignments. The implicit cache option is used to form a join of all previous states at each program structure. (1) provides a modified declaration for **S**. (3) provides the modified definition for the assignment. The join with the previous value associated with the identifier has been removed. The modified definition for the while statement is in (4).

(2) provides the modified definition for the program. As **S** returns the output at the end of the program and is included in the cache, the value of `c` is not required. However, it is used in the let clause to force evaluation of the semantic function for the statement that forms the body of the program. The cache associated with the body of the program contains a map from labels for instances of statements to the local `Cstores`. The unary ∇ operator forms a join of `Cstores` over all labels and provides the required mapping.

(1) **S[[Stat]]** : **Strict** Cstore$_1$ → Cstore$_2$
collect Cstore$_2$;

(2) **P[[(Prog s)]]** = **let** c = **S[[s]]** $\perp_{Cstore}$[ ] **in** ∇[[s]].cache;
—— c *is used to force the evaluation of* **S**

(3) **S[[(Assign** i e)]] =
λx. **let** n = **E[[e]]**x **in** x[n / [[i]]];

(4) **S[[(While** b s)]] =
fix(λf. λx. **let** bt = **B$_t$[[b]]** x **and**
bf = **B$_f$[[b]]** x **in**
(f ((**S[[s]]** bt) ∇ x)) ∇ bf;

Figure 4. Modifications to Figure 3.

3.2.2. Improving efficiency

The *sparse conditional constant* algorithm proposed in [18] improves efficiency by traversing a data structure called the *static single assignment* graph. The structural nature of our framework forces the evaluation of the specification to be equivalent to a traversal of the program flow graph. The incremental nature of the interpreter may provide some improvements in efficiency by avoiding the re-evaluation of portions of the program. However, the use of aggregate values, such as the values from the domain Cstore, as inputs to functions results in unnecessary re-evaluation of many functions. For example, a re-definition of an identifier is considered as a change in the entire Cstore and a function is re-evaluated even if its output does not depend on the value associated with that identifier.

In this section, we describe the use of Def-Use information to avoid such unnecessary evaluations. First, we provide specifications for an analysis that produces Def-Use information and then modify the constant propagation specification in the previous section to use this information to result in a more efficient analysis.

3.2.2.1. Use-Def analysis

As the denotational nature of the framework is more suited to forward-flow analysis techniques than backward-flow analysis, the Def-Use information is obtained through separate passes. The first pass provides Use-Def information and the second inverts it. The specification for obtaining Use-Def information is in Fig. 5.

Domains:
Id, Exp, B-Exp, Stat, Prog = syntactic;
Label_Set = **powerset of** label;
Def_Store = **store** [[Id]] → Label_Set;
UD_Store = **store** label → Label_Set;
Result_Domain = Def_Store × Label_Set;

Function Declarations:
E[[Exp]] : Def_Store → Label_Set;
B[[B-Exp]] : Def_Store → Label_Set;
S[[Stat]] : Def_Store → Result_Domain
collect Result_Domain ↓ 2;
P̂[[Prog]] : → UD_Store;

Function Definitions:
P̂[[(Prog s)]] = **let** x = **S[[s]]** $\perp_{Def\_Store}$[ ] **in** [[s]].cache;
—— x *is used to force evaluation of* **S**

S[[(Assign i e)]] =
λx. **let** L = **E[[e]]**x **in** (x[{$.label} / [[i]]] , L);
S[[(If b s_1 s_2)]] =
λx. **let** Lb = **B[[b]]** x **and**
Ls1 = **S[[s_1]]** x **and** Ls2 = **S[[s_2]]** x **in**
(Ls1↓1 ∇ Ls2↓1 , Lb ∇ Ls1↓2 ∇ Ls2↓2);
S[[(While b s)]] =
fix(λf. λx. **let** Lb = **B[[b]]** x **and**
Ls = **S[[s]]** x **in**
(f Ls↓1) ∇ (x , Lb ∇ Ls↓2);
S[[(Comp s_1 s_2)]] = λx. **let** Ls1 = **S[[s_1]]** x **in**
S[[s_2]] Ls1↓1 ∇ (⊥ , Ls1↓2);

E[[(Id id)]] = λx.x[[[id]]];
E[[(Int_Lit n)]] = λx. $\perp_{Label\_Set}$;
E[[(Add e_1 e_2)]] = λx. (**E[[e_1]]** ∇ **E[[e_2]]**) x;

Figure 5. UD-Chain

The result of the semantic function **S** consists of two components. The first component is a map from the identifiers defined within the statement to the label at which it is defined. The second component is the set of labels of current definition sites for all identifiers used in the statement. The implicit cache collects the latter component and provides a map between each statement and

the definition sites for all identifiers used within the statement. The definitions for **B** are very similar to the one for **E**.

3.2.2.2. Def-Use analysis

The specification to create the Def-Use information from the Use-Def information is provided in Fig. 6. For each statement, **S** inverts the Use-Def store and outputs a Def-Use store.

Domains:

```
Id, Exp, B-Exp, Stat, Prog = syntactic;
Label_Set = powerset of label;
DU_Store = store label --> Label_Set;
UD_Store = store label --> Label_Set;
```

Function Declarations:

```
S[[Stat]] :     UD_Store --> DU_Store;
P̄[[Prog]] :     --> DU_Store;
```

Function Definitions:

```
P̄[[(Prog s)]] = let ud = P̂[[(Prog s)]] in
                    S[[s]] ud;

S[[(Assign i e)]] =
        λx. ⊥[foreach l in x[$.label] {$.label} / l];
S[[(If b s1 s2)]] =
        λx. let du = ⊥[foreach l in x[$.label] {$.label} / l]
            and Ls1 = S[[s1]] x and Ls2 = S[[s2]] x in
                du ∇ Ls1 ∇ Ls2;
S[[(While b s)]] =
        λx. let du = ⊥[foreach l in x[$.label] {$.label} / l]
                and Ls = S[[s]] x in
                        du ∇ Ls;
S[[(Comp s1 s2)]] =
        λx. let du = ⊥[foreach l in x[$.label] {$.label} / l] in
                du ∇ (S[[s2]] ∇ S[[s1]]) x;
```

Figure 6. DU-Chain

3.2.2.3. Using Def-Use information

The modifications to the conditional constant propagation specification are in Fig. 7. The Def-Use store created by the previous specification is passed as an extra argument to the semantic function **S**. For an assignment statement, if an identifier is assigned a value different from the last assignment (if any), then the pre-defined function MustEval is used to mark the labels for the use-sites for evaluation. MustEval returns the second argument and performs the marking as a side-effect. This is the only departure in the framework from a purely functional model.

(a) **S[[Stat]]** :

```
Strict Cstore1 --> DU_Store --> Cstore2
                collect Cstore2;
```

(b) **P̄[[(Prog s)]]** = let du = **P̄[[(Prog s)]]** and
c = **S[[s]]** $\perp_{Cstore}$[] du in ∇[[s]].cache;
—— c *is used to force evaluation of* **S**

(c) **S[[(Assign i e)]]** =
λx. λy. **let** n = **E[[e]]**x **in**
(n=x[[[i]]]) --> x ,
MustEval y[$.label] x[n / [[i]]];

Figure 7. Modifications to use Def-Use information

Initially, all labels are marked for evaluation. The interpreter erases the mark whenever the corresponding semantic function is evaluated. While a label is unmarked, the corresponding semantic function is not re-evaluated even if the input to the function has changed. The output from the previous evaluation is assumed to be still valid. The input is checked with the previous evaluation even for functions corresponding to marked labels since an update may not propagate changes to all marked functions.

3.2.3. Precision and efficiency

The use of Def-Use information improves efficiency without affecting the precision (i.e. the number of constants discovered). Although the Def-Use analysis considers unexecutable paths, only semantic functions corresponding to statements on executable paths are considered for evaluation in the constant propagation analysis. Marking of program parts on unexecutable paths has no effect on the precision. Hence, the objections raised by Wegman and Zadeck in [18] to the use of Def-Use chains do not apply here.

As the semantic functions are monotonic in their argument and the variables can change value only twice, each semantic function is evaluated, in the worst case, 2V+1 times where V is

the number of variables in the program. Hence, the analysis has the asymptotic complexity of $A \times V$ where A is the number of nodes in the abstract syntax tree. With respect to a flow graph model, this translates to a complexity of $E \times V$ where E is the number of edges in the corresponding flow graph. For the language used in our examples, it can be shown that the number of edges in the flow graph is linearly proportional to N, the number of nodes in the flow graph. Hence, this analysis has the same asymptotic complexity as the Sparse Conditional Constant algorithm proposed by Wegman and Zadeck [18].

3.3. Integer Range Analysis

The specification for integer range analysis is shown in Fig. 8. The function definitions for **P** and **S** are the same as for the specification for constant propagation. The domain on which they are defined is, however, different. The definitions for $\tilde{\mathbf{E}}$, $\tilde{\mathbf{B}}$**t** and $\tilde{\mathbf{B}}$**f** can be defined in a fashion similar to the constant propagation specification and have been left out. The goal of this example is to demonstrate the derivation of a specification to incorporate a specialized technique to improve the precision of inference.

As in the conditional constant propagation specification, a lifted domain is used for the program store to avoid collecting information from unreachable paths. The least element of `Int_Range` denotes unknown range information at reachable points. The ordering defined is a partial order in the domain of equivalence classes induced by the ordering. An equivalence class contains all sets of integer bounded by the same two integers (e.g. {1,2,10} and {1,3,5,7,10} belong to the same equivalence class). The equivalence class is represented by the convex-closure which contains all the elements within the two bounds. A formal definition of the ordering can be found in [17].

The semantic function $\tilde{\mathbf{E}}$ approximates the standard semantic function **E** by defining the arithmetic operations over integer ranges. The semantic functions $\tilde{\mathbf{B}}$**t** and $\tilde{\mathbf{B}}$**f** together approximate the standard semantic function **B**. However, they do not return boolean values. Since we would like to make the analysis flow-sensitive and use the information in boolean expressions, they act as filters. $\tilde{\mathbf{B}}$**t** provides an approximation to the input states in which **B** on the same expression would evaluate to *true* while $\tilde{\mathbf{B}}$**f** provides an approximation to the input states in which **B** would evaluate to *false*. The precision of such an approximation depends on the nature of the expression and the complexity of the algorithm used. In the worst case, both $\tilde{\mathbf{B}}$**t** and $\tilde{\mathbf{B}}$**f** are identity functions.

Domains:

```
Id, Exp, B-Exp, Stat, Prog = syntactic;
Int_Range = powerset of integer
            ordered by range_order
            represented by convex-closure;
Istore = lifted store [[Id]] --> Int_Range;
```

Function Declarations:

```
E[[Exp]] :      Strict Istore --> Int_Range;
Bt[[B-Exp]] :   Strict Istore --> Istore;
Bf[[B-Exp]] :   Strict Istore --> Istore;
S[[Stat]] :     Strict Istore1 --> Istore2
                       collect Istore1;
P[[Prog]] :     --> Istore;
```

Function Definitions:

```
P[[(Prog s)]] = let r = S[[s]] ⊥ [] in (∇ [[s]].cache) ∇ r;

S[[(Assign i e)]] =
        λx. let n = E[[e]]x in x[n / [[i]]];
S[[(If b s1 s2)]] =
        λx. let bt = Bt[[b]] x and bf = Bf[[b]] x in
                S[[s1]] bt ∇ S[[s2]] bf;
S[[(While b s)]] =
        fix(λf. λx. let bt = Bt[[b]] x and
                        bf = Bf[[b]] x in
                    (f ((S[[s]] bt) ∇ x)) ∇ bf;
S[[(Comp s1 s2)]] = λx. (S[[s2]] ∘ S[[s1]]) x;
```

Figure 8. Integer Range Analysis

3.3.1. Increasing precision

The analysis in this specification is similar in precision of inference to the one described in [2]. However, the use of "independent" attributes method [7] results in some crude approximations. For example, consider the two code fragments below:

(a) `a:=1;while a<4 do a:=a + a od`

(b) `a:=1;b:=1;while a<4 do a:=a + 1;b:=b + 1 od`

The value of `a` at the end of the code segment (a) will be approximated by the range `[4,6]`, while the value of `b` at the end of the code segment (b) will be approximated by `[1,∞]`. Although the approximation in (a) may not look particularly poor, the

approximation in (b) is not very useful. Rather than using the inefficient "relational" attribute method [7], we provide a variation that increases the precision of the range inference at the cost of slightly decreased efficiency compared to the original specification.

The imprecision in the above examples arises from the use of the join operator in the semantic equation for the while loop to ensure termination. The join operator introduces imprecision since it takes the least upper bound of two approximations which, in general, results in the introduction of some states that may never occur in the standard interpretation. One could avoid this by keeping every range that occurs separate. However, from an implementation standpoint this would be very inefficient. The gain in precision is totally offset by the space requirements and the computational costs involved in evaluating the semantic functions separately over each of the ranges.

The difference in the specifications for the standard and approximation semantics for the while loop suggests a compromise solution. In the standard semantics, the boolean condition in the while loop is evaluated for each possible state at the beginning of the loop. In the approximation semantics, the boolean condition is evaluated on the least upper bound of all the previous approximations to the state at the beginning of the loop. We will approximate the sequence of approximations that occur at a point with two values. The intuitive interpretation for the two values is that the first value is the approximation corresponding to the most recent evaluation while the second is the least upper bound of all the previous approximations at that point. We use the observation that the second value is always available in the cache.

To express this in our formalism, we will use the domain

`Int_Tuple = Int_Range × Int_Range`

rather than just `Int_Range`. The modified semantic equation for the while loop is given in Fig. 9. The rest of the semantic equations are modified slightly to compute over a pair of integer ranges. The required information is obtained by using the cache associated with the current evaluation instance of the semantic function $\bar{S}$ to get the state at the label corresponding to the syntactic object s. The value in the cache always lags one evaluation behind and is used to collect the approximations of the previous evaluations. For simplicity, we have assumed in this modification, that programs do not have nested loops. In the presence of nested loops we can either bound the depth of nesting to some level d and use a domain of cross-product of d+1 ranges or evaluate the fixed point separately for each element in the tuple.

This modified analysis provides the very sharp approximations `a : [4,4]` and `b : [4,4]` at the end of the example code segments above.

```
S[[(While b s)]] =
    fix(λf. λx. let bt = B_t[[b]] x and
                bf = B_f[[b]] x and
                c = $.cache[[s]].label] in
                 f (S[[s]]( bt ∇ (⊥,(c↓1 ∇ c↓2))) ∇ bf;
```

Figure 9. Modifications to increase precision

3.4. Collecting interpretation of expressions

A model for a collecting interpretation for functional languages has been developed by Hudak and Young in [5]. This model was designed to avoid the use of power domains by effectively separating the specification of an analysis and the specification for the collection of information from the analysis. We demonstrate below how the implicit caching feature of our framework can be adopted to express this model. The specification for collecting applicative order (call-by-value) semantics of a first-order functional language whose abstract syntax is in Fig. 10 is given in Fig. 11. For brevity, the specification uses some notation (e.g. ellipses) that is not currently supported by the framework. They can be easily added to the framework.

$x \in Bv$	Bound variables
$f \in Fv$	Function Variables
$k,p \in Con$	Constants
$e \in Exp$	Expressions
$pr \in Prog$	Programs

$p ::= \{e; f_i(x_1, ..., x_n) = e_i\}$

$e ::= k$
$\quad | \ x$
$\quad | \ p(e_1, ..., e_n)$
$\quad | \ f(e_1, ..., e_n)$

Figure 10. Abstract Syntax

Domains:

```
exp, prog, id, con : syntactic;
values : lifted (integer + boolean);
functions : function values* --> values
                merge cache;
VarStore : store [[id]] --> values;
FuncStore : recursive store [[id]] --> functions;
```

Function Declarations:

```
E[[exp]] : VarStore --> FuncStore --> values
        collect powerset of values;
P[[prog]] : --> values;
K[[con]] : values* --> values;
```

Function Definitions:

```
E[[k]] = λvs.λfs. K[[k]];
E[[x]] = λvs.λfs. vs[[x]];
E[[p(e1 ..., en)]] =
   λvs.λfs. K[[p]] E[[e1]]vs fs ..., E[[en]]vs fs;
E[[f(e1 ..., en)]] =
   λvs.λfs. fs[[f]] E[[e1]]vs fs ..., E[[en]]vs fs;

P[[e; fi(x1 ..., xn) = ei]] =
   let fs = ⊥[strict(λ(d1, .... dn). E[[ei]] ⊥ [dj/xj] fs)/fi]
       and v = fix cache E[[e]] ⊥ fs
         in [[e]].cache;

K[[if]] = λb.λt.λf. b --> t,f
   {b --> cache$1 ∇ cache$2 , cache$1 ∇ cache$3};
K[[+]]  = λd1.λd2. Add d1 d2
   {cache$1 ∇ cache$2};
```

Figure 11. Collecting Interpretation

4. A prototype implementation

A prototype for the framework has been implemented for use in conjunction with the Synthesizer Generator [14]. The specification of an analysis can be coupled with a synthesizer specification to generate an editor that performs that analysis. The framework was implemented mostly in SSL, an applicative language supported by the synthesizer generator. It consists of about 1500 lines of source code of which roughly 350 lines constitute the interpreter for the specifications. The rest of the code provides the support features that may be used for specific analysis techniques. The simple and concise nature of the interpreter eases formal verification of the consistency of the interpreter with the axiomatic definition of the denotational specification language.

The denotational functions are maintained as attributes in the abstract syntax tree. The incremental evaluation scheme available in the synthesizer generator [3] aids in the construction of the denotational function corresponding to the program. The evaluation of the function itself cannot be expressed in the attribute framework since the fixed-point evaluation would result in cyclic attributes that are not currently supported by the synthesizer generator. The interpreter, written as a function in SSL, maintains its own cache of input and output values for each of the semantic functions. This cache is used to avoid re-evaluation of semantic functions for the same inputs.

The improved range analysis specification provided in the previous section has been incorporated in an editor for a subset of Pascal that provides automatic declaration of variables with subrange types specified for integer variables. The complexity of the analysis varies linearly with the number of statements in the program in the worst case. However, the complexity of evaluation of fixed-point solutions for loop constructs can increase exponentially in the number of nested loops in the worst case. This does not pose a problem in analyses where the length of any non-decreasing chain in the domain of approximations is bounded by a small number. In range analysis, we trade off some precision and use a domain that bounds the length of any chain which essentially determines the number of times a loop must be unwound before the most general approximation is made. We use a bound of 50 for the generated editor.

5. Summary

We have developed and implemented a framework that can be used to construct concise high-level specifications of program analysis techniques. Use of such a framework in a system such as the Synthesizer Generator allows program analysis techniques to be incorporated into program development environments without a need to supply implementation details. This aids in rapid development of new program analysis techniques as well as techniques that share common features but are customized for specific applications. The choice of a denotational framework to express these specifications allows formal proofs of correctness to be established for each of these analysis techniques. The facilities provided by the framework result in clear and concise specifications that aid in the understanding of the corresponding analysis techniques.

References

1. A. W. Appel, "Semantics-Directed Code Generation," pp. 315-324 in *Proc. 12th ACM Symp. on Principles of Programming Languages*, (January 1985).

2. P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," pp. 238-252 in *Proc. 4th ACM Symp. on Principles of Programming Languages*, (January 1977).

3. Allan Demers, Thomas Reps, and Tim Teitelbaum, "Incremental evaluation for attribute grammars with application to syntax-directed editors.," pp. 105-116 in *Proc. 8th ACM Symp. on Principles of Programming Languages*, (1981).

4. P. Hudak, "A semantic model of reference counting and its abstraction," pp. 45-62 in *Abstract Interpretation of Declarative Languages*, ed. S. Abramsky, C. Hankin,Ellis Horwood, West Sussex (1987).

5. P. Hudak and J. Young, "A Collecting Interpretation of Expressions (Without Powerdomains)," pp. 107-118 in *Proc. 15th ACM Symp. on Principles of Programming Languages*, (January 1988).

6. J. Hughes, "Analysing strictness by abstract interpretation of continuations," pp. 63-102 in *Abstract Interpretation of Declarative Languages*, ed. S. Abramsky, C. Hankin,Ellis Horwood, West Sussex (1987).

7. N. D. Jones and S. S. Muchnik, "Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra," pp. 380-393 in *Program flow analysis: Theory and applications*, Prentice-Hall (1981).

8. N. D. Jones and A. Mycroft, "Data flow analysis of applicative programs using minimal function graphs.," pp. 296-306 in *Proc. 13th ACM Symp. on Principles of Programming Languages*, (January 1986).

9. G. A. Kildall, "A Unified Approach to Global Program Optimization," *ACM Symp. on Principles of Programming Languages*, pp. 194-206 (1973).

10. F. Nielson, "A denotational framework for data flow analysis," *Acta Informatica* **18** pp. 265-287 (1982).

11. A. Pal, "Generating Execution Facilities for Integrated Programming Environments," Tech. Report 676, University of Wisconsin-Madison (1986). Ph.D. thesis

12. U. F. Pleban, "Compiler Prototyping Using Formal Semantics," pp. 94-105 in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, , Montreal, Canada (June 1984).

13. W. W. Pugh, "Incremental computation and the incremental evaluation of function programs," Tech. Report 88-936, Cornell University (1988). Ph.D. thesis

14. T. Reps and T. Teitelbaum, *The Synthesizer Generator Reference Manual*, Springer-Verlag, New York (Third Edition, 1988).

15. R. Sethi, "Control Flow Aspects of Semantics-Directed Compiling," *ACM Transactions on Programming Languages and Systems* **5**(4)(1983).

16. G. A. Venkatesh and C. N. Fischer, "A framework for denotational specification of program analysis techniques," Tech. Report in preparation, University of Wisconsin-Madison (1989).

17. G. A. Venkatesh and C. N. Fischer, "Construction of program analysis techniques for use in program development environments," Tech. Report #811, University of Wisconsin-Madison (1989).

18. M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," Tech. Report #CS-88-02, Brown University (1988).