CONSTRUCTION OF PROGRAM ANALYSIS TECHNIQUES FOR USE IN PROGRAM DEVELOPMENT ENVIRONMENTS

G A Venkatesh and Charles N. Fischer

Computer Sciences Technical Report #811

January 1989

CONSTRUCTION OF PROGRAM ANALYSIS TECHNIQUES FOR USE IN PROGRAM DEVELOPMENT ENVIRONMENTS

G A Venkatesh and Charles N. Fischer

Computer Sciences Technical Report #811

January 1989

Construction of program analysis techniques for use in program development environments

G A Venkatesh and Charles N. Fischer
University of Wisconsin - Madison
venky@cs.wisc.edu, fischer@cs.wisc.edu

Abstract

Program analysis techniques have been used in the past to aid in translation of programs. Recently, techniques have been developed to aid in the construction of programs. Use of such techniques in interactive program synthesizers result in effective program development environments. 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.

This report provides an overview of a framework that has been designed to facilitate the construction of program analysis techniques for use in programming environments generated by a tool such as the Synthesizer Generator [17]. The framework supports high-level specifications of analysis techniques in a denotational fashion where the implementation details can be ignored. Many of the features commonly required in program analysis techniques are provided as primitives in the framework resulting in clear and concise specifications that aid in the understanding of the corresponding analysis techniques. The framework is exemplified by specifications for dependence analysis and scalar range analysis for imperative programs.

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

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

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 effective program synthesis environments.

The construction of programming environments is made easier through systems such as the Synthesizer Generator [17]. Programming environments are automatically generated from formal language definitions. The Synthesizer Generator provides a formalism based on acyclic attribute grammars that can be used to incorporate several analysis techniques (e.g. type inference) into the generated environments. An efficient evaluator is provided for attribute equations. This allows analysis techniques to be developed through high-level specifications whose implementation details can be ignored. Since most flow-analysis techniques result in cyclic attribute equations, there has been research [9, 21] to provide efficient evaluation of cyclic attribute equations.

1.1. Semantic aspects of analysis

The seminal paper on *abstract interpretation* [3] introduced the notion that a wide variety of program analysis techniques could be specified formally as interpretations over abstract domains. The formal specifications would allow the construction of proofs 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 are designed to terminate for any program. It is useful to provide formal reasoning that guarantees termination.

(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.

1.2. Denotational Specifications

Although the original work on abstract interpretation involved operational specifications for flowchart languages, most of the later studies based on abstract interpretation [6, 8, 11, 13, 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. The consistency with standard semantics can be established easily with respect to the denotational semantics of the language. Studies such as [1, 15, 16, 19] demonstrate the possibility of providing efficient evaluators for denotational specifications.

1.3. Problems with denotational frameworks

The use of denotational frameworks for specification of program analysis techniques tends to be less than ideal for several reasons. First, the denotational functions make assertions 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 [7, 13].

This problem is analogous to the use of copy attributes in attribute grammar frameworks to propagate information between nodes in the abstract syntax tree. This results in messy specifications that are difficult to understand. It also requires a wasteful duplication of effort in maintaining the "cache" parameter. Moreover, this method 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.

A major difference between the construction of standard semantics and alternate semantics arises from the fact that there is no single alternate semantics that is the most appropriate for all applications. In program analysis techniques, there is always a trade-off between the effort involved in the analysis and the precision with which information is obtained from the analysis. One can construct a sequence of analyses that infer the same property but to varying degrees of precision corresponding to varying efficiency of implementation. An application would choose the implementations that is most appropriate.

The approach in [14] is to provide a single denotational specification on which several interpretations (including the precise standard interpretation) are defined. However, we find that the development of certain efficient and/or precise analysis techniques require carefully crafted specifications (The specification

for scalar range analysis described later in the paper is an example). Designing a single specification on which different interpretations result in different analysis techniques leads to difficulties in construction of the specification as well as understanding of the analysis techniques. Moreover, as the result of the analysis depends on the interpretation used, the proofs for correctness for each analysis must use a formalism for the interpretation as well.

1.4. Solutions

We have developed 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 specifications for a program 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 the same for all specifications. This interpretation model may be optimized to include particularly efficient evaluation techniques such as incremental evaluation. The correctness of the interpretation model is proved once, independent of any particular specification.

1.5. Organization of the report

Section 2 presents an incomplete description of the framework that is sufficient for the example specifications. A complete description will be provided in a following report. Section 3 provides specifications for dependence analysis and scalar range analysis for imperative programs. The formalism for establishing the correctness of the range analysis technique can be found in Appendix B. Range analysis technique for a richer language and the correctness results will be provided in a following report. Some implementation details for the framework are outlined in Section 4. Section 5 discusses ongoing research and Section 6 summarizes the report.

2. The framework

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 Specification

Domains

primitive domain	::=	integer I boolean I ordinal I syntactic I label
domain	::=	*primitive domain*
	|	*domain* × *domain*
	|	*domain* + *domain*
	|	**lifted** *domain*
	|	**powerset of** *domain*
	|	**Store** *domain* —> *domain*

Ordering

primitive ordering	::=	subset I flat I arithmetic
ordering	::=	*primitive ordering* | `logical expression`

Representative

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

All the primitive domains except *ordinal* are flat domains. *ordinal* has a total ordering induced by the arithmetic $\leq$ relation. The lifted domain introduces a special element $\perp$ as the least element into the base domain. Since an ordering definition may not produce 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 an implementation. It is used in the specification for range analysis.

A *store* is used commonly enough in specifications to warrant a pre-defined facility in the framework. Access and update operations are provided by the framework. We plan to provide more than one implementation for the *store* so that a specification can select the implementation that is optimal for that specification. It is also possible for the user to define a *store* through auxiliary functions.

2.2. Function specification

Type specification

function type	::=	*domain* < —> *domain* >* < **collect** *domain*$_i$ >

Function constructors

primitive function	::=	Identity | Bottom
function	::=	*primitive function*
	|	*function* ∘ *function*
	|	*function* ∇ *function*
	|	*function* —> *function* , *function*
	|	λx.*function* x
	|	**fix** *function*

The **collect** option defines the values to be associated with the labels corresponding to the syntactic objects for which the function was defined. Such a function can be assumed to have an extra parameter of the domain type **store** *label* —> *domain*$_i$. The function constructors are defined using a formalism based on axiomatic rules. This provides a formal definition of the collection of results of partial evaluations which can be used to reason about the correctness of the intermediate states. Any evaluation scheme used to provide an interpreter for the specifications must be consistent with these definitions. The choice of an axiomatic system allows such a formal definition while avoiding over-specification of possible evaluation schemes. We provide a sample of such definitions below. The complete set of definitions will be provided in a following report.

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

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

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

These definitions constitute an operational definition for the specification language. They can be intuitively understood as the numerator containing the operational steps necessary to evaluate the denominator. There is no evaluation order specified for the terms in the numerator. 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) 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 union (∇) operator for functions when the collect option is used.

3. Example specifications

We provide two specifications for the small language defined below. The first specification is for a simple data dependency analysis. The analysis assumes that all paths are possible. The second specification is for scalar range analysis (integer range analysis in the example) that avoids certain paths that are not possible. The range analysis technique was motivated by a research goal to extend type inferencing to automatically select type implementations for a given program.

Type inferencing has been proposed in a variety of contexts. It is an integral part of some language definitions, most notably ML [4]. Its use in programming environments has also been explored [2]. A program development environment can use type inferencing to automatically synthesize type definitions for variables used in a program. However, there may be more than one implementation of a type that satisfies the specification of the type (e.g., alternative implementations of a stack). In a given program, one implementation may be a better choice than another in terms of space or time efficiency (or even correctness if the specification is not complete). Proper selection of an implementation can provide valuable assistance to programmers.

Typically, type inference is done through a static analysis of the program [12]. Careful selection of a type or its implementation, however, requires information about the dynamic properties of the program. Execution of the program on sample inputs will, in most cases, be insufficient. Moreover, possible non-termination of some of these executions make this solution unsuitable for use in program synthesis environments. We need static analysis techniques that can provide useful information about the dynamic behavior of the program over all possible inputs.

Range analysis is an example of a technique that is required for inference of type implementations. Imperative languages rely heavily on the notion of states and transformations between states. Information about the states that could occur during the execution of any program in an imperative language is necessary to determine whether a particular type implementation is appropriate. A state can be approximated by

providing approximations to the values that the variables in the program assume during actual executions. For scalar variables, the approximations are provided as scalar ranges.

For range analysis to be useful in inference of type implementations, the analysis should provide sufficiently precise approximations to actual executions to avoid choosing the most general implementation in all cases. However, the techniques used to make the analysis terminate and/or efficient may result in poor approximations. For example, consider the following code segments:

(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**

Using the "independent" attribute method [10] for state descriptors, an abstract interpretation [3] will result in the value of `a` at the end of the code segment (a) being approximated by the range `[4..6]`, while the value of `b` at the end of the code segment (b) is approximated by `[1..∞]`. Although the approximation in (a) may not look particularly poor, the approximation in (b) is not very useful. The use of the "relational" attribute method [10] to increase the precision of approximation will result in an extremely inefficient analysis. We have developed an improved technique to increase the precision of range inference without serious impact on efficiency. Our technique would determine a range of `[4..4]` for all three variables upon exit from the loops.

3.1. Abstract Syntax and Standard Semantics

For the illustration we will use a small language that will be sufficient to demonstrate the various issues. 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 and the syntax for boolean expressions. The boolean expressions consist of the standard primitive boolean operators. A program consists of a sequence of statements. The abstract syntax and standard semantics can be found in Fig. 1 and Fig. 2 respectively.

3.2. Dependence Analysis

The specification for dependence analysis is shown in Fig. 3. The semantic function **E** provides a set of labels where the identifiers used in the expression were last defined. The semantic function **S** provides an updated store if an identifier was declared in that statement as well as the set of all labels where the identifiers used in the statement were last defined. The **collect** option is used to collect the latter values and associate them with labels corresponding to statements. The proof of correctness for this specification is straightforward and will be provided in a later report. However, note the close correspondence between the semantic function definitions for standard semantics and the alternate semantics.

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 s_1 s_2) | Conditional Statement

(**While** b s) | Loop statement

(**Comp** s_1 s_2) Statement Composition

e ::= (**Id** id) |

(**Add** e_1 e_2)

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* s_1 s_2)]]	=	λx. (**B**[[b]] —> **S**[[s_1]] , **S**[[s_2]]) x
S[[(**While** *b s*)]]	=	*fix*(λf. λx. (**B**[[b]] —> (f ∘ **S**[[s]]) , Identity) x)
S[[(**Comp** s_1 s_2)]]	=	λx. (**S**[[s_2]] ∘ **S**[[s_1]]) x
E[[(**Id** *i*)]]	=	λx.x[i]
E[[(**Add** e_1 e_2)]]	=	λx.Add **E**[[e_1]]x **E**[[e_2]]x

Figure 2. Standard Semantics

3.3. Scalar Range Analysis

The specification is shown in Fig. 4. Although the powerset constructor provides the empty set as the least element, we use a lifted powerset to detect unreachable paths. The least element represents program state at unreachable points while the empty set 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. The formal definition for this approximation domain is in Appendix B.

Domains:

Id, Exp, B-Exp, Stat, Prog	=	syntactic
Label_set	=	**powerset of** label **ordered by** subset
Prog_Store	=	**Store** ⟦Id⟧ → label

Function Declarations:

E⟦Exp⟧	=	Prog_Store → Label_set
B⟦B-Exp⟧	=	Prog_Store → Label_set
S⟦Stat⟧	=	Prog_Store → Prog_Store → Label_set **collect** Label_set
P⟦Prog⟧	=	Prog_Store → Prog_Store → Label_set **collect** Label_set

Function Definitions:

P⟦(**Prog** s)⟧	=	λx. **S**⟦s⟧x
S⟦(**Assign** i e)⟧	=	λx.(x[$.label / i] , **E**⟦e⟧x)
S⟦(**If** b s_1 s_2)⟧	=	λx. (**S**⟦s_1⟧ ∇ **S**⟦s_2⟧ ∘ **B**⟦b⟧)x
S⟦(**While** b s)⟧	=	*fix* (λf. λx. (((f ∘ **S**⟦s⟧) ∇ Identity) ∘ **B**⟦b⟧)x)
S⟦(**Comp** s_1 s_2)⟧	=	λx. (**S**⟦s_2⟧ ∘ **S**⟦s_1⟧) x
E⟦(**Id** i)⟧	=	λx.{x[i]}
E⟦(**Add** e_1 e_2)⟧	=	λx. (**E**⟦e_1⟧ ∇ **E**⟦e_2⟧) x

Figure 3. Dependence Analysis

The definitions for $\bar{\mathbf{E}}$, $\bar{\mathbf{B}}\mathbf{t}$ and $\bar{\mathbf{B}}\mathbf{f}$ will be included in a following report on range analysis. The semantic function $\bar{\mathbf{E}}$ approximates the standard semantic function **E** by defining the arithmetic operations over integer ranges. The semantic functions $\bar{\mathbf{B}}\mathbf{t}$ and $\bar{\mathbf{B}}\mathbf{f}$ 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. $\bar{\mathbf{B}}\mathbf{t}$ provides an approximation to the input states in which **B** on the same expression would evaluate to *true* while $\bar{\mathbf{B}}\mathbf{f}$ 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 $\bar{\mathbf{B}}\mathbf{t}$ and $\bar{\mathbf{B}}\mathbf{f}$ are identity functions. This has the same effect as the traditional data-flow analysis algorithms that assume all paths are always taken.

To highlight the improvements in conciseness and clarity afforded by our framework, contrast the specification in Fig. 4 with a specification of the same analysis in Appendix A constructed using standard

---

Domains:

Int_Range = **powerset of** integer **ordered by** range_order **represented by** convex-closure
Approx_Range = **lifted** Int_Range
Id, Exp, B-Exp, Stat, Prog = syntactic
Prog_Store = **Store** ⟦Id⟧ –> Approx_Range

Function Declarations:

$\bar{\mathbf{E}}$⟦Exp⟧	=	Prog_Store –> Approx_Range
$\bar{\mathbf{B}}\mathbf{t}$⟦B-Exp⟧	=	Prog_Store –> Prog_store
$\bar{\mathbf{B}}\mathbf{f}$⟦B-Exp⟧	=	Prog_Store –> Prog_store
$\bar{\mathbf{S}}$⟦Stat⟧	=	$Prog_Store_1$ –> $Prog_Store_2$ **collect** $Prog_Store_1$
$\bar{\mathbf{P}}$⟦Prog⟧	=	$Prog_Store_1$ –> $Prog_Store_2$ **collect** $Prog_Store_1$

Function Definitions:

$\bar{\mathbf{P}}$⟦(**Prog** s)⟧	=	λx. $\bar{\mathbf{S}}$⟦s⟧x
$\bar{\mathbf{S}}$⟦(**Assign** i e)⟧	=	λx.(x[($\bar{\mathbf{E}}$⟦e⟧x) / i]
$\bar{\mathbf{S}}$⟦(**If** b s_1 s_2)⟧	=	λx. (($\bar{\mathbf{S}}$⟦s_1⟧ ∘ $\bar{\mathbf{B}}\mathbf{t}$⟦b⟧) ∇ ($\bar{\mathbf{S}}$⟦$s_2$⟧ ∘ $\bar{\mathbf{B}}\mathbf{f}$⟦b⟧))x
$\bar{\mathbf{S}}$⟦(**While** b s)⟧	=	*fix* (λf. λx. ((f ∘ (($\bar{\mathbf{S}}$⟦s⟧ ∘ $\bar{\mathbf{B}}\mathbf{t}$⟦b⟧) ∇ Identity)) ∇ $\bar{\mathbf{B}}\mathbf{f}$⟦b⟧) x)
$\bar{\mathbf{S}}$⟦(**Comp** s_1 s_2)⟧	=	λx. ($\bar{\mathbf{S}}$⟦s_2⟧ ∘ $\bar{\mathbf{S}}$⟦s_1⟧) x

Figure 4. Integer Range Analysis

---

denotational formalism.

3.3.1. Increasing Precision

At the beginning of Section 3 we provided two example code segments for which the specification produces rather poor approximations. The imprecision in both is due to the use of the union operator in the semantic equation for the while loop to ensure termination.

The union 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 stand-point 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

$$\texttt{Approx_Range} = \textbf{lifted}\ \texttt{Int_Range} \times \texttt{Int_Range}$$

The modified semantic equation for the while loop is given in Fig. 5. The rest of the semantic equations remain essentially the same. The required information is obtained by using the cache associated with the current evaluation instance of the semantic function $\bar{\mathbf{S}}$ 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. The complete modified specification will be available in a following report on range analysis.

This modified analysis provides the very sharp approximations `a : [4,4]` and `b : [4,4]` at the end of code segments at the beginning of section 3. The improvement in precision possible using this technique (without serious impact on efficiency) makes it highly practical compared to range analysis

$$\tilde{\mathbf{S}}[\![(\mathbf{While}\ \mathtt{b}\ \ \mathtt{s})]\!] = \mathit{fix}(\lambda f.\ \lambda x.\ ((f \circ ((\tilde{\mathbf{S}}[\![\mathtt{s}]\!] \circ \tilde{\mathbf{B}}\mathbf{t}[\![\mathtt{b}]\!]) \nabla \mathtt{Collect})) \nabla \tilde{\mathbf{B}}\mathbf{f}[\![\mathtt{b}]\!])\ x)$$

where $\mathtt{Collect} = \lambda x.\ (\bot\ ,\ c{\downarrow}1\ \nabla\ c{\downarrow}2)$

and $c = \$.\mathrm{cache}[\tilde{\mathbf{S}}[\![\mathtt{s}]\!].\mathrm{label}]$

Figure 5.

examples found in literature.

4. A Prototype Implementation

A prototype of our framework has been implemented using SSL, an applicative language supported by the synthesizer generator. It consists of about 1000 lines of source code of which roughly 200 lines constitute the interpreter for denotational specifications. The rest of the code provides the support features 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 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 evaluations result in cyclic attributes which are not 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 determine whether re-evaluation is required when the program is edited.

The improved range analysis specification provided in the previous section has been incorporated in an editor for a small subset of Pascal. It 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 may be unwound before the most general approximation is made. We use a bound of 50 for the generated editor.

5. Continuing research

Further research is concentrated on two areas:

(1) Testing the adequacy of the support provided by the framework when applied to various analysis and to richer languages.

(2) Providing efficient incremental evaluators for denotational specifications.

As mentioned in Section 1.3, there is no single alternate semantics that is appropriate for all applications. Even if language features do not add any significant complexity to the construction of a standard or alternate semantics, they introduce alternatives in the design of program analysis techniques. For example, consider the problem of selecting type implementations for local data objects of a procedure. One option is to select an implementation that is suitable for calls from all call-sites. However, if the context from each call-site is kept separate, it may be possible to select more efficient implementations that are suitable for each particular call-site rather than the most general implementation. Further, an analysis that distinguishes different call instances from the same call-site can lead to even more efficient selections. This precision is obtained at the cost of increased analysis effort. Since any one of these three decisions may be appropriate depending on the application, it should be possible to specify any or all of them.

To allow such specifications, the framework should be able to support spatial and temporal differentiation of domain values. Spatial differentiation can be supported easily through a labeling mechanism. This labeling mechanism is very similar to the one currently used to distinguish different instances of denotational functions that appear in an abstract syntax tree. However, temporal differentiation is not so straightforward since an approximate program analysis usually does not create temporal instances in one-one correspondence with the standard interpretation.

The usefulness of the framework also depends on the efficiency of an interpreter for the specifications. As one of the main goals of designing our framework is to allow program analysis techniques to be incorporated into interactive program synthesizers, it is useful to design an evaluator that is capable of incremental computations that reflect incremental modifications to a program. Although the interpreter currently provided with the framework maintains a cache of input and output values for each of the functions to decide whether a re-evaluation is required, this facility is not sufficient for fixed-point computations. The intermediate results of a fixed-point computation in a previous evaluation are not available during re-evaluation. Even if the inputs to a fixed-point computation have changed, the re-evaluation may repeat some of the intermediate computations in the previous evaluation. For example, consider the following code segment:

(*a*) `a := 1;` **while** `a<10` **do** `a := a + 1` **od**

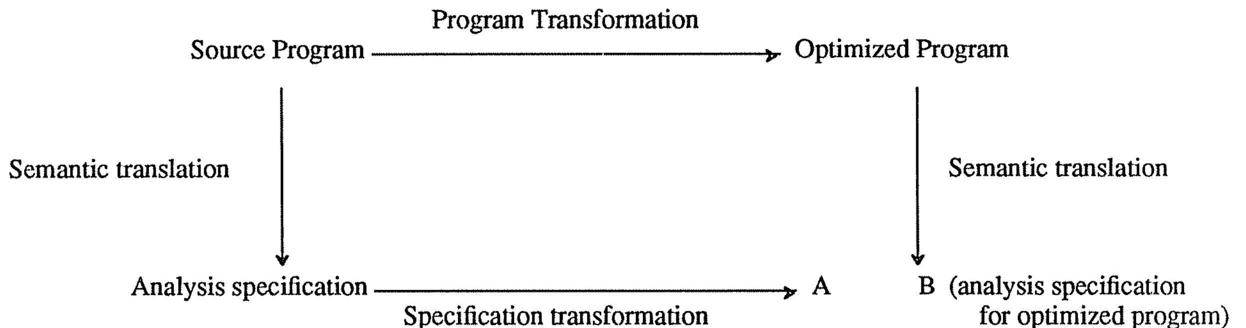
Suppose that a range analysis had been carried out on the above code segment and the code segment was later modified in the following two ways:

(*b*) `a := 5;` **while** `a<10` **do** `a := a + 1` **od**

(*c*) `a := 1;` **while** `a<10` **do** `a := a + 2` **od**

Consider the problem of recomputing the fixed-point solution to the function corresponding to the while loop. In (b), only the input to the function has changed while in (c), the function itself has changed slightly. In both cases, the re-computation is very similar to the original computation. We are designing an incremental fixed-point evaluation algorithm that can use information from previous evaluations to handle small perturbations in input values and/or the functional specification.

The axiomatic definition for the denotational specification language determine the possible standard interpreters for the language. These definitions are adequate when the interpreters are simple. However, if the standard interpretation consists of a translator that performs certain optimizing code transformations, the information obtained by a program analysis specification may no longer be valid if the transformations are not reflected in the program analysis. This is essentially the same problem noted by Hennessy [5] in debugging optimized programs. If one could provide transformation rules for the specification language that correspond to the transformation rules in the source language, a specification for an optimized program could be automatically obtained from the specification for an un-optimized program.



In standard interpretation, it is necessary to ensure that the specification A is "equivalent" to B. In program analysis specification however, it is sufficient to show that A provides a "safer" approximation than B. Our goal is to identify the transformations that would make such a derivation possible.

6. Summary

We have developed and implemented a framework that can be used to construct 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.

7. Appendix A - Range analysis specification in standard denotational formalism

Domains:

```
Prog_Store = Id --> Approx_Range
Cache = Label --> Prog_Store
Env = Prog_Store × Cache
```

Auxiliary functions:

$\bot_{\texttt{Prog_Store}} = \bot[\bot_{\texttt{Approx_Range}} / \texttt{l}]$ for all $\texttt{l} \in$ Label
$\nabla_c : \texttt{Prog_Store}^2 \rightarrow \texttt{Prog_Store} = \lambda x_1.\lambda x_2.\lambda i.\ x_1(i) \nabla x_2(i).$

$\bot_{\texttt{Cache}} : \texttt{Cache} = \lambda l.\ \bot_{\texttt{Prog_Store}}$
$+ : \texttt{Cache} \rightarrow \texttt{Cache} \rightarrow \texttt{Cache} = \lambda c_1.\lambda c_2.\lambda l.\ c_1(l) \nabla_c c_2(l)$
$\texttt{update} : \texttt{Cache} \rightarrow \texttt{Label} \rightarrow \texttt{Prog_Store} \rightarrow \texttt{Cache} = \lambda c.\lambda l.\lambda a.\ c + \bot[a/l]$

$\texttt{Stick} : \texttt{Env} \rightarrow \texttt{Env} \rightarrow \texttt{Env} = \lambda e_1.\lambda e_2.(e_1{\downarrow}1\ ,\ e_1{\downarrow}2 + \texttt{update}\ \ e_2{\downarrow}2\ \$.label\ \ e_2{\downarrow}1)$
$\nabla_s : \texttt{Env} \rightarrow \texttt{Env} \rightarrow \texttt{Env} = \lambda e_1.\lambda e_2.(e_1{\downarrow}1 \nabla_c e_2{\downarrow}1\ ,\ e_1{\downarrow}2 + e_2{\downarrow}2)$

Semantic Function Declarations:

$\bar{\mathbf{E}}$: Exp --> Prog_Store --> Approx_Range
$\bar{\mathbf{Bt}}$: B-Exp --> Prog_Store --> Prog_Store
$\bar{\mathbf{Bf}}$: B-Exp --> Prog_Store --> Prog_Store
$\bar{\mathbf{S}}$: Stat --> Env --> Env
$\bar{\mathbf{P}}$: Prog --> Env --> Env

Semantic Function Definitions:

$\bar{\mathbf{P}}[\![(\textbf{Prog}\ \texttt{s})]\!] = \lambda a.\lambda c.\ \bar{\mathbf{S}}[\![\texttt{s}]\!]a\ c$

$\bar{\mathbf{S}}[\![(\textbf{Assign}\ \texttt{i}\ \ \texttt{e})]\!] = \lambda a.\lambda c.\ \texttt{Stick}\ (a[(\bar{\mathbf{E}}[\![\texttt{e}]\!]a)/\texttt{i}]\ ,\ \bot_{\texttt{Cache}})\ a\ c$

$\bar{\mathbf{S}}[\![(\textbf{If}\ \texttt{b}\ \texttt{s}_1\ \texttt{s}_2)]\!] = \lambda a.\lambda c.\ \texttt{Stick}\ ((\bar{\mathbf{S}}[\![\texttt{s}_1]\!](\mathbf{Bt}[\![\texttt{b}]\!]a)\ c) \nabla_s (\bar{\mathbf{S}}[\![\texttt{s}_2]\!](\mathbf{Bf}[\![\texttt{b}]\!]a)\ c))\ a\ c$

$\bar{\mathbf{S}}[\![(\textbf{While}\ \texttt{b}\ \ \texttt{s})]\!] = \lambda a.\lambda c.\ \texttt{Stick}\ u\ a\ c$

where $u = fix(\lambda f.\ \lambda a'.\lambda c'.\ f\ ((\bar{\mathbf{S}}[\![\texttt{s}]\!](\mathbf{Bt}[\![\texttt{b}]\!]a)\ c') \nabla_s (a', \bot_{\texttt{Cache}})) \nabla_s (\mathbf{Bf}[\![\texttt{b}]\!]a), \bot_{\texttt{Cache}}))a\ c$

$\bar{\mathbf{S}}[\![(\textbf{Comp}\ \texttt{s}_1\ \texttt{s}_2)]\!] = \lambda a.\lambda c.Stick\ (\bar{\mathbf{S}}[\![\texttt{s}_2]\!](\bar{\mathbf{S}}[\![\texttt{s}_1]\!]a\ c))\ a\ c$

8. Appendix B - Formalism to express correctness of range analysis

To demonstrate the correctness of the analysis, we must show that the analysis terminates for any program and that it provides approximations that are safe and consistent with the standard semantics of the program. To reason about the correctness of the approximations at each program point, we require a definition for standard interpretation that establishes the intermediate states that occur whenever control reaches that program point. This definition is provided as a "sticky" (or collecting) [14] standard semantics expressed in our framework. This specification is in Figure 6.

The cache holds a sequence (possibly unbounded) of integers for each identifier.An update to the store consists of pushing the value into the beginning of the sequence. Note that any implementation would need to keep just the head of the sequence at any time. However, we will assume the existence of the sequence to reason about the correctness of the range analysis. The definitions for **E** and **B** remain the

---

Domains:

```
Int_Sequence = lifted sequence of integer
L_Integer = lifted integer
L_Bool = lifted boolean
Id, Exp, B-Exp, Stat, Prog = syntactic
Prog_Store = Store [[Id]] --> Int_Sequence
```

Function Declarations:

$$\mathbf{E}[\![\texttt{Exp}]\!] = \texttt{Prog_Store} \longrightarrow \texttt{L_Integer}$$
$$\mathbf{B}[\![\texttt{B-Exp}]\!] = \texttt{Prog_Store} \longrightarrow \texttt{L_Bool}$$
$$\bar{\mathbf{S}}[\![\texttt{Stat}]\!] = \texttt{Prog_Store}_1 \longrightarrow \texttt{Prog_Store}_2 \ \textbf{collect}\ \texttt{Prog_Store}_1$$
$$\bar{\mathbf{P}}[\![\texttt{Prog}]\!] = \texttt{Prog_Store}_1 \longrightarrow \texttt{Prog_Store}_2 \ \textbf{collect}\ \texttt{Prog_Store}_1$$

Function Definitions:

$$\bar{\mathbf{P}}[\![(\textbf{Prog}\ s)]\!] = \lambda x.\ \bar{\mathbf{S}}[\![s]\!]x$$
$$\bar{\mathbf{S}}[\![(\textbf{Assign}\ i\ e)]\!] = \lambda x.(\ x[(\mathbf{E}[\![e]\!]x :: x[i]) / i]$$
$$\bar{\mathbf{S}}[\![(\textbf{If}\ b\ s_1\ s_2)]\!] = \lambda x.\ (\mathbf{B}[\![b]\!] \longrightarrow \bar{\mathbf{S}}[\![s_1]\!]\ ,\ \bar{\mathbf{S}}[\![s_2]\!])\ x$$
$$\bar{\mathbf{S}}[\![(\textbf{While}\ b\ s)]\!] = \mathit{fix}(\lambda f.\ \lambda x.\ (\mathbf{B}[\![b]\!] \longrightarrow (f \circ \bar{\mathbf{S}}[\![s]\!])\ ,\ \text{Identity})\ x)$$
$$\bar{\mathbf{S}}[\![(\textbf{Comp}\ s_1\ s_2)]\!] = \lambda x.\ (\bar{\mathbf{S}}[\![s_2]\!] \circ \bar{\mathbf{S}}[\![s_1]\!])\ x$$

Figure 6. Sticky Standard Semantics

---

same. The following lemma establishes the consistency of this specification with the standard semantics in Figure 2.

Lemma 1 : *For any program* p, *both* $\mathbf{P}[\![p]\!]\perp$ *and* $\bar{\mathbf{P}}[\![p]\!]\perp$ *diverge or* $\mathbf{P}[\![p]\!]\perp = \lambda i.\mathrm{Head}(\bar{\mathbf{P}}[\![p]\!]\perp(i))$.

Since the correctness of the range analysis is proved with respect to the collecting standard semantics, the results obtained from the analysis is meaningful in only those implementations of the language that are consistent with the collecting semantics specifications.

8.1. Approximation domain

We provide a formal definition for an approximation domain for integer range analysis from which the domain used in Figure 4 is obtained as a special case. Since variables of only integer types are allowed in the example, the domain of denotable values must contain sets of integers as elements. This naturally suggests a power domain construction. Although, there have been criticisms against the use of power domains [7], the construction required for our purpose is quite simple. We require only a discrete power domain construction for the illustration. Unlike in the case of functional programs, the extension of our techniques to include higher-order programs is not a critical issue for imperative programs. The use of formal procedures is not central to programming in imperative languages. Hence we do not consider the problems that may occur in extending our techniques to include formal procedures as a serious limitation.

Out of the several power domain orderings available [18,20], we use a variant of the Hoare ordering.

Definition 1: *Let D be a cpo with ordering* $\leq_D$. *The Hoare ordering* $\mathbf{H} \subseteq \mathbf{P}(D)^2$ *is a relation defined as*

$$A \leq B \;\; iff \;\; \forall a \in A, \exists b \in B,\; a \leq_D b$$

If D is a flat domain, the ordering can be rephrased as:

$$A \leq B \;\; iff \;\; A \subseteq (B \cup \{\perp_D\})$$

This ordering conforms to the decision that non-termination (represented by $\perp_D$) as a possible solution is ignored (although we preserve the ability to detect unreachable paths as discussed later).

The ordering in a power domain is constructed so as to preserve both the "information content" ordering in the base domain as well as the the set approximation ordering. For approximation semantics we may need to to force additional "information content" orderings among the power domain elements. An example is designating groups of elements in the power domain as equivalent in "information content" and coercing them into equivalence classes. We show below such a construction for the base domain of integers.

The construction is motivated by the application in which a set of integers is approximated by an integer range. The semantics of operations over integers can then be efficiently approximated by operations

using range arithmetic.

Definition 2: *The Convex Closure of every subset $A \subseteq D$ of the base domain D with ordering $\leq_D$ is the set $C_D(A)$ defined as*

$$C_D(A) = \{ b \in D \mid \exists a,c \in A.\ a \leq_D b \leq_D c \}.$$

The set A is called Convex in D iff $A = C_D(A)$

Definition 3: *Let Int_f be the flat domain of integers and Int_t the domain of integers with total ordering defined by the arithmetic relation $\leq$, $\forall a \in Z.\ a \leq a+1$.*

$\mathbf{P}(Int_f)$ is the power domain whose elements are the subsets of the proper (non-$\perp$) elements of Int_f pre-ordered by the relation $\mathbf{H_c} \subseteq \mathbf{P}(Int_f)^2$, denoted by $\leq_c$ and defined as

$$A \leq_c B \quad iff \quad \forall a \in A.\ a \in C_{Int_t}(B).$$

$\mathbf{P}(Int_f)/\leq_c$ is the quotient of $\mathbf{P}(Int_f)$ with respect to $\leq_c$. The elements of $\mathbf{P}(Int_f)$ are grouped into equivalence classes using the equivalence relation $\equiv_c$ defined as

$$A \equiv_c B \quad iff \quad A \leq_c B \text{ and } B \leq_c A.$$

We denote by $\mathbf{P}_r(Int_f)$, the domain of these equivalence classes with the partial ordering $\leq_c$.

Lemma 2: *For any set $A \in \mathbf{P}(Int_f)$, the equivalence class containing A, denoted by [A], contains an unique Convex (in Int_t) set denoted by Rep([A]) such that*

$$Rep([A]) = C_{Int_t}(P) \ \forall P \in [A].$$

We designate this element as the representative for the equivalence class.

Lemma 3: *[{}] is the least element of $\mathbf{P}_r(Int_f)$.*

Definition 4: *The "collecting" operator ∇: $\mathbf{P}_r(Int_f)^2 \rightarrow \mathbf{P}_r(Int_f)$ is defined as*

$$[A] \nabla [B] = [A \cup B].$$

Lemma 4: ∇ *is well defined and continuous.*

Although the loss of information in using a covering range may seem reasonable in many cases, it may seriously affect the precision of approximation in cases where a variable assumes values in disjoint ranges that are wide apart. It is reasonable to have implementations that keep ranges apart (not necessarily disjoint) although the number of such ranges maintained for each identifier is usually fixed for a given implementation. To accommodate this approach we define further refinements to our domain as follows:

Definition 5: *The sequence of domains $\mathbf{P}_r(Int_f)^n$ for finite $n \in \mathbf{N}$, $n > 0$ is obtained by forming the n-ary cross products of $\mathbf{P}_r(Int_f)$.*

The pre-ordering $\leq_c^n \subseteq \mathbf{P}_r(Int_f)^{n\,2}$ is defined as

$$A \leq_c^n B \quad iff \quad \forall i \ \ 1 \leq i \leq n \ \ \exists j \ \ 1 \leq j \leq n \ \ \forall \alpha \in Rep(A \downarrow i).\ \alpha \in Rep(B \downarrow j)$$

Definition 6: *$Aint^n$ is the domain $(\mathbf{P}_r(Int_f)^n/\leq_c^n)_\perp$, the quotient of $\mathbf{P}_r(Int_f)^n$ with respect to $\leq_c^n$ lifted by the least element $\perp$ and partially ordered by $\leq_c^n$.*

Definition 7: *∇^n denotes the "collecting" operator ∇ distributed over the cross product.*

The decision to use a lifted domain for $Aint^n$ is motivated by the requirement that unreachable program points be detected. Hence even if a single range is used in an implementation, it is preferable to use $Aint^1$(`Approx_Range` in Figure 4) rather than $\mathbf{P}_r(Int_f)$. The least element $\perp \in Aint^1$ for a program point denotes that the program point is as yet unreachable while the element $[\{\}] \in Aint^1$ (we will represent it by $\perp\!\!\!\perp$) denotes that the program point is reachable while the range is still unknown. As an illustration, the interpretation framework described in [3] using a domain equivalent to $\mathbf{P}_r(Int_f)$ would infer the range of `a` at the end of the code segment

```
a := 1; while false do a := 5 od;
```

as `[1,5]` while with the use of $Aint^1$ it is possible to infer the range as `[1,1]` in the same framework.

8.2. Correctness results

We now provide the formalism required to express the correctness of the range analysis. The proofs will be included in a following report on range analysis. However we have provided some comments where necessary to get an intuitive understanding.

Definition 8: *The Approximation function Ψ : Store —> A-Store is defined as*

$$\Psi = \lambda s.\ \perp[(\gamma_n(s(i)))/i]\ \forall i \in Id$$

where $\gamma_n : Int_f \rightarrow Aint^n$ is

$$\gamma_n = \lambda x.[\rho(x) \times \rho(\perp)^{n-1}]$$

and $\rho : Int_f \rightarrow \mathbf{P}_r(Int_f)$ is

$$\rho = \lambda x.\ \text{if}\ x = \perp\ then\ [\{\}]\ else\ [\{x\}].$$

Definition 9: *The pointwise ordering $\leq_s : A\text{-}Store \rightarrow A\text{-}Store$ is defined as*

$$s_1 \leq_s s_2\ \ iff\ \ \forall i \in Id.\ s_1(i) \leq_c^n s_2(i).$$

We assume that $\tilde{\mathbf{E}}$ and $\tilde{\mathbf{B}}$ have been defined so that the following propositions are true:

Proposition 1 : *The definition of $\tilde{\mathbf{B}}$ satisfies the following conditions:*

(1) *For all $b \in B\text{-}Exp$ and $s \in A\text{-}Store$, $\tilde{\mathbf{B}}[\![b]\!]s$ is effectively computable.*

(2) *For all $b \in B\text{-}Exp$ $\tilde{\mathbf{B}}[\![b]\!]$ is monotonic.*

(3) *$\tilde{\mathbf{B}}$ safely approximates $\mathbf{B}$:*

$$\forall b \in B\text{-}Exp\ \text{if}\ \mathbf{B}[\![b]\!]s\ \ then\ \Psi(s) \leq_s \tilde{\mathbf{B}}[\![b]\!]s' \downarrow 1\ \ else\ \Psi(s) \leq_s \tilde{\mathbf{B}}[\![b]\!]s' \downarrow 2$$

$$for\ all\ s' \in A\text{-}Store\ \ such\ that\ \Psi(s) \leq_s s'$$

Proposition 2 : *The definition of $\tilde{\mathbf{E}}$ satisfies the following condition:*

(1) *For all* $e \in Exp$ *and* $s \in A\text{-}Store$, $\tilde{\mathbf{E}}[\![b]\!]s$ *is effectively computable.*

(2) *For all* $e \in Exp$ $\tilde{\mathbf{E}}[\![e]\!]$ *is monotonic.*

(3) $\tilde{\mathbf{E}}$ *safely approximates* $\mathbf{E}$*:*

$$\forall e \in Exp \quad \Psi(\mathbf{E}[\![e]\!]s) \leq_s \tilde{\mathbf{E}}[\![e]\!]s' \text{ ,for all } s' \in A\text{-}Store \text{ such that } \Psi(s) \leq_s s'$$

Lemma 5 : *The definition of* $\tilde{\mathbf{S}}$ *in Fig. 2 satisfies the following conditions:*

(1) *If the domain A-Store is of finite length, then for all* $s \in Stat$ *and* $a \in A\text{-}Store$, $\tilde{\mathbf{S}}[\![s]\!]a$ *is effectively computable.*

(2) *For all* $s \in Stat$ $\tilde{\mathbf{S}}[\![s]\!]$ *is monotonic.*

(3) $\tilde{\mathbf{S}}$ *safely approximates* $\bar{\mathbf{S}}$*:*

$$\forall s \in Stat \quad \Psi(\bar{\mathbf{S}}[\![s]\!]a) \leq_s \tilde{\mathbf{S}}[\![s]\!]a' \text{ ,for all } a' \in A\text{-}Store \text{ such that } \Psi(a) \leq_s a'$$

Proof: The finite length assumption is satisfied in implementations where the domain Int_f is finite or an arbitrary bound is imposed on any chain in $Aint^n$ due to efficiency reasons. In the latter case the limit of any chain above a fixed length is an element that safely approximates all states. The proofs for the assignment, conditional and composition are straight-forward. The fixed point meaning for the loop statement can be represented by **lub**(f_0a, f_1a, ...), a $\in$ *A-Store* where each of the subfunctions f_i is non-recursive and effectively computable. Using the observation that the input to the recursive call in the fixed point equation is always greater than the input to the current activation as well as the finite length pre-condition one can show that there exists a finite k such that $f_i a = f_k a$ for all $i > k$. Hence the fixed point meaning is the least upper bound of values of a finite set of effectively computable functions and is therefore effectively computable. (2) and (3) are proved using fixed point induction.

Lemma 6 : *For any program* p, *if* $C_s = \bar{\mathbf{P}}[\![p]\!].cache$ *and* $C_r = \tilde{\mathbf{P}}[\![p]\!].cache$ *then for all* $l \in Label$ *and for all* $i \in Id$,

$$\forall x \in C_s\ l\ i,\ \gamma_n\ x \leq C_r\ l\ i.$$

(γ_n is defined in definition 8).

Proof: Using the axiomatic definitions that define the collecting operation it can be shown that for every evaluation of a collecting standard semantic function, there is an evaluation of the corresponding range analysis semantic function such that the input to the range analysis semantic function safely approximates the input to the standard semantic function. This guarantees that an approximation to every program state that occurs at each program point in the evaluation of the standard semantics is added to the cache.

Lemmas 5 and 6 lead directly to the following correctness result for the range analysis specification:

Theorem 1: *The range analysis specification is correct with respect to the standard semantics, that is, the range analysis specification for any program in the given language is effectively computable and provides safe approximations that are consistent with the standard semantics for the language.*

Similar results hold for the improved range analysis specification. In addition, the following result can also be proved.

Theorem 2: *The range analysis with modifications suggested in Section 3.3.1 is more precise than the analysis of Fig. 4.*

Proof: It is proved by showing that the modified semantics can never produce an approximation that is less precise and that there are examples (such as (a) and (b) in Section 3) where it provides more precise approximations.

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. R. Conway, D. DeJohn, and S. Worona, "A User's Guide to The COPE Programming Environment," TR 84-599, Department of Computer Science, Cornell University (April 1984).

3. 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).

4. M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF*, Springer-Verlag, Berlin (1979).

5. J. Hennessy, "Symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems* **4**(3) pp. 323-344 (1982).

6. 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).

7. 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).

8. 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).

9. L. G. Jones and J. Simon, "Hierarchical VLSI design systems based on attribute grammars," pp. 58-69 in *Proc. 13th ACM Symp. on Principles of Programming Languages*, (January 1986).

10. 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).

11. 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).

12. M. Kaplan and J.D. Ullman, "A general scheme for the automatic inference of variable types," pp. 60-75 in *Proc. 5th ACM Symp. on Principles of Programming Languages*, (1978).

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

14. F. Nielson, "Towards a denotational theory of abstract interpretation," pp. 219-245 in *Abstract Interpretation of Declarative Languages*, ed. S. Abramsky, C. Hankin,Ellis Horwood, West Sussex

(1987).

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

16. 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).

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

18. D. A. Schmidt, *Denotational Semantics - A methodology for language development*, Allyn and Bacon, Boston (1986).

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

20. H. Sondergaard and P.Sestoft, "Non-Determinacy and its semantics," Report NR: 86/12, DIKU, Denmark (January 1987).

21. J. Walz and G. F. Johnson, "Incremental evaluation for a general class of circular attribute grammars.," *Sigplan Notices* **23**(7) pp. 209-221 (July 1988). Proc. of the Sigplan '88 Conf. on Programming Language Design and Implementation