Models and Interpretations: Formalizing Multiple Semantics [Extended Abstract]

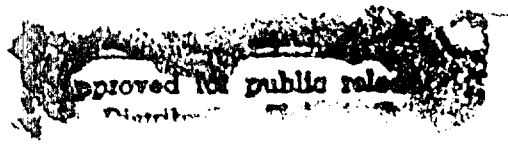
Michael Karr and Steve Rozen

—Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138
mike@soi.com

December 1, 1989

Abstract

We often think of a program as having different meanings in different contexts. When inferring types, for example, a compiler computes an upper bound on the possible types of each expression during execution. These upper bounds can be thought of as an interpretation of the program. When writing numerical software a programmer sometimes reasons as if the program operated on mathematical real numbers but at other times reasons about numbers with only finite precision, two rather different interpretations. This paper defines formal notions of *interpretation* and *model* to provide a counterpart to our intuitive understanding of the meanings of a program. Briefly, an interpretation is an assignment of labels to program terms, subject to constraints imposed by a model. This formalization lets us reason rigorously about relationships between different meanings and provides a conceptual framework for designing algorithms for program analysis, transformation, and execution. We develop and motivate the details through the example of a model in which an interpretation is a program's (interprocedural) flow graph. We also briefly discuss several other models in which interpretations are computed as part of the transformation and compilation machinery of the E-L programming environment and language.



This work was supported in part with funds provided by the Defense Advanced Research Projects Agency and the Naval Ocean Systems Center under contract N00014-85-C-0710 with the Office of Naval Research.

1 Introduction

How many meanings should a program have? We may be tempted to reply "one," but there is a sense in which, as programmer and compiler work on a program, it takes on many meanings. For example, when inferring types, the compiler is construing the program to have a certain meaning, and when running the program under a symbolic debugger, the programmer is working with another idea of the meaning of the program. Neither of these meanings is the same as what we usually think of as "the" meaning of a program, the values computed when a program is executed.

We would like to reason formally about these various meanings and about relationships between meanings. For example, we would like to say that the "meaning" of an expression in type analysis is an upper bound on the set of possible types of the expression during any execution. We would also like a rigorous foundation for designing algorithms to compute the various "meanings" we use when analyzing, transforming, and executing programs.

1.1 The Language

Our discussion requires us to apply the concepts of model and interpretation to an example language. For this we use a simplified form of Base E-L, the lambda-calculus-based intermediate language of the E-L System. Henceforth, "Base E-L" refers to this idealized Base E-L and "program" and "term" refer to a Base E-L program.

Base E-L's terminals are formed from two sets, C and N, of *constants* and *parameter names* (*names* for short). Examples of constants are "built-in" functions such as `+` and values such as Boolean `false`. We write $(t_1\ t_2)$ to denote an *application* of t_1 to t_2; t_1 is the operator and t_2 is the argument in the application. A name, n, is introduced in an *abstraction*, written $\lambda n[t]$; n may occur in any sub-term of t. We call t the *body* of the abstraction. When an abstraction, $\lambda n[t]$, is the operator at an application $(t_1\ t_2)$, n has the value produced by t_2.

2 Models and Interpretations

Our formalism depends on two constructs: (i) *interpretations*, which are, roughly speaking, mappings from program terms to labels that represent "meaning", and (ii) *models*, which give the set of labels and define relations that constrain interpretations. We want to formulate these constructs so that a model can embody any reasonable semantics. For example, we would like to be able to specify non-deterministic order of evaluation of operator and argument at an application or normal order, instead of applicative order, semantics.

We develop the notions of model and interpretation in the context of a specific example, an interprocedural flow-graph model; we also refer to a formal execution model that space limitations prevent us from fully developing.

A flow graph is a representation of a program in which vertices contain computations (e.g. instructions) and arcs represent the flow of control between computations. An interpretation in a flow-graph model assigns a flow graph to a program. An interpretation in an execution

model assigns sets of states to terms, and prescribes relations on pairs of sets. An execution is then formalized as a chain of related states. Intuitively, a flow-graph interpretation is correct with respect to an execution model if there is a path in the flow-graph interpretation for every possible chain in the execution model.

2.1 Labelings

A labeling is a map from each sub-term in a term to elements which contain *labels*. For each model, labels must capture the semantic contribution of a term. As a result, the definition of the set of labels, L, varies from model to model. For flow-graph models, L is the set of all vertices in all possible (finite) flow graphs. Thus a flow-graph labeling maps terms to pieces of flow graphs.

We formally define a labeling as follows. Let T_c, T_n, T_a, and T_b be the sets of Base E-L constant, name, application, and abstraction terms, respectively. Let

$$\begin{aligned} \mathcal{I}_c &: T_c \rightarrow L \times C \times L, \\ \mathcal{I}_n &: T_n \rightarrow L \times N \times L, \\ \mathcal{I}_a &: T_a \rightarrow L \times Ap \times L, \text{ and} \\ \mathcal{I}_b &: T_b \rightarrow L \times Ab \times L. \end{aligned}$$

Then the map $\mathcal{I} = \mathcal{I}_c \cup \mathcal{I}_n \cup \mathcal{I}_a \cup \mathcal{I}_b$ is a labeling. If $\mathcal{I}(t) = \langle l, x, l' \rangle$, we call l the *pre-label* and l' the *post-label* of t.

We define a set, Ap, of *application surrogates*, and a set, Ab, of *abstraction surrogates* of a model. Their purpose is to distill information from the application or abstraction to help constrain interpretations; like L, the sets Ap and Ab vary from model to model. For example, in flow-graph models, Ap encodes the set of values that, in any possible execution, may be the operator at the application. Flow-graph models use this information to compute all possible flow paths when an application is performed, e.g. into any abstraction that may act as the application's operator.

2.2 Model Relations

Some latitude in the specification of a model comes from the ability to define L, Ab, and Ap to suit the model's particular requirements. In addition, the specification of a model relies on the definition of certain relations on L, Ab, Ap, C, and N. We define an *interpretation in a model* to be a labeling that satisfies the constraints imposed by the model's relations. In a flow-graph model we want the relations to ensure that the vertices in an interpretation are connected in a way (possibly via additional vertices not in the interpretation) that reflects the flow of control implied by an execution model.

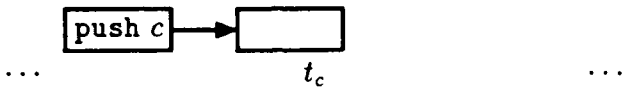
2.2.1 Interpretations of Constant, Name, and Abstraction Terms

A model uses the relation $cnst \subseteq L \times C \times L$ to constrain the interpretation of constant terms. Let t_c be a constant term containing the constant c. For a labeling, $\mathcal{I}$, to be an interpretation, we require that $\mathcal{I}(t_c) = \langle l, c, l' \rangle \in cnst$.

For flow-graph models, *cnst* is defined to be the set of all triples $\langle v, c, v' \rangle$ such that

- the contents of v is **push** c, and
- there is an arc from v to v', the out-degree of v is 1, and the in-degree of v' is 1. (When these three conditions obtain we write $v \rightarrow v'$.)

We can depict this as follows:

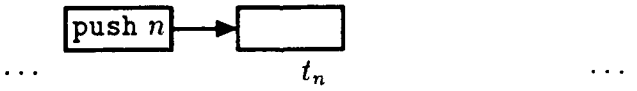


Our definition of *cnst* constrains neither the in-degree of v, nor the out-degree and contents of v'; in flow-graph models these attributes are constrained by relations for the enclosing term.

The relation $name \subseteq L \times N \times L$ works analogously for Base E-L name terms. Let t_n be a name term containing the name n. For a labeling, $\mathcal{I}$, to be an interpretation, we require that $\mathcal{I}(t_n) = \langle l, n, l' \rangle \in name$. For flow-graph models, we define *name* to be the set of all triples $\langle v, n, v' \rangle$ such that

- the contents of v is push n, and
- $v \rightarrow v'$.

The following depicts the interpretation of t_n.



The relation $abst \subseteq L \times Ab \times L$ constrains the interpretation of abstraction terms. Let $t_b = \lambda n[t]$ be an abstraction term. For a labeling, $\mathcal{I}$, to be an interpretation we require that $\mathcal{I}(t_b) \in abst$. In the flow-graph model *abst* is defined to be the set of triples $\langle v, b, v' \rangle$ such that

- the contents of v is **push-abst** v_{t_b}, where v_{t_b} is the first vertex of the flow graph for t_b, and
- $v \rightarrow v'$.

The following depicts the flow-graph interpretation of t_b.

Recall that $b \in Ab$ is an abstraction surrogate. Note that we use v_{t_b} in defining *abst*. To make v_{t_b} available to *abst*, in flow-graph models we define Ab to be $L \times N$, where for $b = \langle v_{t_b}, n \rangle \in Ab$, v_{t_b} is the first vertex in the graph of the abstraction term t_b, and n is the parameter name of the abstraction; n is used below in defining the contents of v_{t_b}.

We have now described how models constrain interpretations of constant, name, and abstraction terms. We still need mechanisms to constrain interpretations of abstraction bodies and of applications, i.e., intuitively, to specify what an abstraction body and what an application mean.

2.2.2 Constraining Interpretations of Abstraction Bodies

The relations *abst-entry* $\subseteq L \times Ab$ and *abst-exit* $\subseteq L \times Ab$ constrain interpretations at the left and the right sides, respectively, of abstraction bodies.

Let t_b be an abstraction term. For a labeling, $\mathcal{I}$, to be an interpretation we require that whenever $\mathcal{I}(t_b) = \langle l, b, l' \rangle$,

- $\exists \langle l'', b \rangle \in$ *abst-entry* such that l'' is the pre-label on the body of t_b, and
- $\exists \langle l''', b \rangle \in$ *abst-exit* such that l''' is the post-label on the body of t_b.

In flow-graph models, *abst-entry* is defined to be the set of pairs $\langle v'', \langle v_{t_b}, n \rangle \rangle$ such that

- the contents of v_{t_b} is **entry** n, where n is the parameter name of t_b, and
- $v_{t_b} \rightarrow v''$.

We call v_{t_b} the *entry vertex of* t_b.

In flow-graph models, *abst-exit* is defined to be the set of pairs $\langle v''', \langle v_{t_b}, n \rangle \rangle$ such that

- v''' has contents **exit**.

We call v''' the *exit vertex of* t_b. The following diagram depicts the labels at the left and right sides of an abstraction body.

2.2.3 Interpretations of Applications

In every model, the relation *app-strt* $\subseteq L \times Ap \times L$ relates an application's pre-label to the operator's pre-label and the relation *app-fnsh* $\subseteq L \times Ap \times L$ relates the argument's post-label to the application's post-label. Let t_a be the application $(t_1\ t_2)$. For a labeling, $\mathcal{I}$, to be an interpretation, we require that whenever $\mathcal{I}(t_a) = \langle l, a, l' \rangle$,

- $\exists \langle l, a, l_l \rangle \in$ *app-strt* such that l_l is the pre-label of t_1,
- the post-label of t_1 is the pre-label of t_2, and

- $\exists\langle l'_l, a, l'\rangle \in$ *app-fnsh* such that l'_l is the post-label of t_2.

The middle requirement is called the continuity constraint; it captures the idea that nothing happens between terms.

Intuitively, control flows from the application's pre-label to the pre-label of the application's operator. Therefore, for flow-graph models we define *app-strt* to be the set of triples $\langle v, a, v_l\rangle$ such that

- $v \rightarrow v_l$, and
- v contains `strt`.

The following depicts the contribution of *app-strt* to the interpretation of an application.

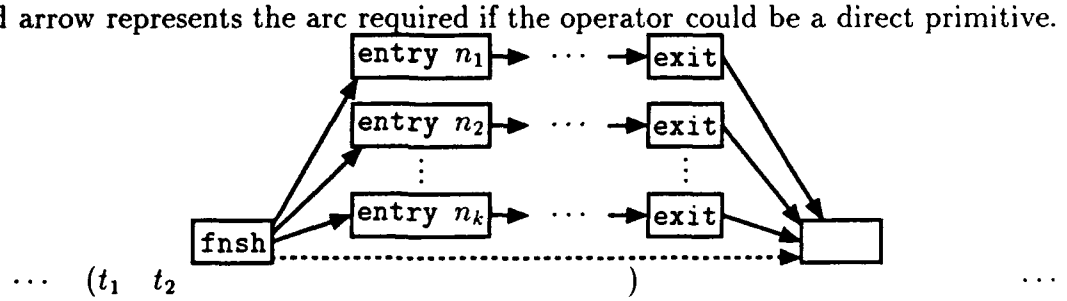
Finally, we must define *app-fnsh* to constrain the graph structure at the right-hand side of an application. In an execution, this is the point at which the values of the operator and argument are known, and control may flow to different sorts of vertices, depending on the operator.

- For primitives such as `+`, control flows from the argument's post-label directly to the post-label of the application. Such primitives are called *direct* primitives.
- For abstractions, control flows from the argument's post-label to the abstraction's entry vertex and from the abstraction's exit vertex to the post-label of the application.

Recall that for flow-graph models the application surrogate, $a \in Ap$, of an application, t_a, encodes the set of all operators that, in any execution, may be the operator at t_a. With this information we can define *app-fnsh* to be the set of all triples $\langle v'_l, a, v'\rangle$ such that

- whenever a contains a direct primitive there is an arc from v'_l to v',
- for every abstraction term, t_b, that can act as the operator at t_a there is an arc from v'_l to the entry vertex of t_b and an arc from the exit vertex of t_b to v', and
- v'_l has no other out-arcs.

The following depicts *app-fnsh*'s contribution to the interpretation of an application. The dashed arrow represents the arc required if the operator could be a direct primitive.



3 Conclusion

In addition to the flow-graph example presented here, the authors have implemented algorithms computing interpretations in several other models as part of the E-L System. These are

- a *dynamic cross reference (DXRF) model*, in which interpretations provide upper bounds on the set of values that may be the operator and argument at an application, as needed for the flow-graph model's application surrogates, and
- a *mode model*, in which interpretations perform a role similar to static type analysis.

The DXRF and flow-graph models translate programs to a form suitable for input to an optimizing code generator based on execution statistics, and the mode model is used in the translation from the surface language to Base E-L.

To summarize, we have presented formal notions of model and interpretation that allow us to ascribe multiple semantics to programs. A model captures the intuitive concept of meaning by defining a set of labels, L, and a map, called an interpretation, from terms to labels. An interpretation must satisfy the model relations.

The formalization we developed helps us state and prove relations between models — for example, that there is a path in a flow graph interpretation for every chain of related states in an execution. Finally, the model formalism has led to a clean design for algorithms implementing the transformation and execution mechanism of the E-L System.