Static inference of properties of applicative programs

Prateek Mishra and **Robert M.Keller**
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract

An applicative program denotes a function mapping values from some domain to some range. *Abstract interpretation* of applicative programs involves using the standard denotation to describe an abstract function from a "simplified" domain to a "simplified" range, such that computation of the abstract function is effective and yields some information, such as type information, about the standard denotation. We develop a general framework for a restricted class of abstract interpretations that deal with non–strict functions defined on non–flat domains. As a consequence, we can develop inference schemes for a large and useful class of functional programs, including functions defined on *streams*. We describe several practical problems and solve them using abstract interpretation. These include inferring *minor signatures* and *relevant clauses* of functions, which have arisen out of our work on a strongly–typed applicative language.

This work was supported in part by a University of Utah Research Fellowship, and grants from IBM, NSF (no. MCS–8106177), and the Defense Advanced Research Projects Agency, US Department of Defense (contract No. MDA903–81–C–0414).

1. Introduction

1.1. Abstract Interpretation of applicative programs

Static inference techniques for applicative programs are of use in developing optimized implementations of applicative languages and enhancing the reliability of applicative programs by deducing aspects of the "type" of a function. The technique of *abstract interpretation* [5, 6] forms the theoretical basis for our techniques. An applicative program denotes a function mapping values from some domain to some range.

Abstract interpretation of applicative programs consists of using the standard denotation to describe an abstract function from a simplified domain ("simplified" in that only particular properties of interest are captured) to a simplified range. The computation of the abstract function yields some information about the standard denotation.

The techniques developed here can cope with a very general class of first–order applicative programs: *non–strict functions acting on non–flat domains*. There is great utility in being able to deal with this class, as many useful applicative programs consist of non–strict functions acting on streams (a non–flat domain) [8, 12]. We formulate a set of simple conditions characterizing a well–formed abstract interpretation, and describe a computational scheme for arriving at the "abstract" denotation of an expression.

Static inference for applicative languages often involves computing a description of values that might be bound to parameters of functions, as well as a description of values returned by functions. Constructing the description as a *set* of values, as is the case for imperative languages [5] is unsatisfactory. Non–strict functions may yield useful results on being applied to expressions whose evaluation does not terminate. Hence the description may need to take termination into account [19]. To do so consistently forces us to consider sets with an order–structure of the kind familiar in work on indeterminacy [21]. Further, in dealing with non–flat domains, as we do, the sets have a more complex order structure than is the case with flat domains, somewhat independent of whether or not termination is of interest.

The standard denotation of a function definition f is written [[f]]:D –> R. The use of *abstraction* functions, which "simplify" the domain and range of a function, are central to our approach. *Abstraction* maps $Abs1$:D –> $A1$ and $Abs2$:R –> $A2$, capture aspects of the domain D and range R in "simpler" domains $A1$ and $A2$ (for a description of "acceptable" abstraction maps see section 3). Clearly, $A1$ and $A2$ should capture some characteristic of the domain (D) and range (R) which are of interest. We then interpret [[f]] as a function, either from $A1$ to $A2$, or from more complex domains constructed out of $A1$ and $A2$. Depending on the structure of the underlying domains ($A1$ and $A2$), several such interpretations may be available. We describe one such interpretation below; a complete discussion may be found in Section 3.

The *union-based* interpretation interprets $[[f]]$ as a function from the *powerdomain* of *A1* (written PD[A1]) to PD[A2]. While [19] has earlier given a similar construction for flat[1] domains, a number of difficulties arise in developing such a construction for non-flat domains [22, 2]. In this paper, we confine our work to using a standard powerdomain construction which is well behaved for *finite A1* and *A2*. Many, but not all, computationally effective abstract interpretations involve such domains. As our examples involve only finite *A1* and *A2* the construction yields the necessary theoretical framework.

The following diagram summarizes the *union-based* interpretation. A function $[[f]]$: *D* -> *R* induces a relation $R_{[[f]]}$ from *A1* to *A2* which in turn induces a function from PD[A1] -> PD[A2]; the function *i* is an embedding function (i:j = {j}). The union-based interpretation for $[[f]]$ is written $[[f]]^{ub}$.

$$\begin{tikzcd}
D \arrow[r, "{[[f]]}"] \arrow[d, "Abs\,1"] & R \arrow[d, "Abs\,2"] \\
A1 \arrow[r, "R_{[[f]]}"] \arrow[d, "i"] & A2 \arrow[d, "i"] \\
PD[A1] \arrow[r, "{[[f]]^{ub}}"] & PD[A2]
\end{tikzcd}$$

$\forall\, a \epsilon A1$,
$a\, R_{[[f]]}\, z, \forall\, z \epsilon \{ Abs2{:}[[f]]{:}x \mid Abs1{:}x = a \}$

$\forall\, A \epsilon PD[A1]$,
$[[f]]^{ub}{:}A = \{ b \mid a R_{[[f]]} b, \forall\, a \epsilon A \}$

Figure I: Union-based interpretation

The action of $[[f]]^{ub}$ on any AϵPD[A1], consists of taking the union of all possible values given by applying [[f]] to the pre-image of A in D; these values are mapped to a point in PD[A2] by *Abs2*. In practice, $[[f]]^{ub}$ cannot be directly computed. Instead an approximation, written $[[f^{ub}]]$, derived from analyzing the representation (function definition) for f is used. As is well known [5] computation of $[[f^{ub}]]$ is only possible if the domain PD[A1] -> PD[A2] possesses the *finite chain* property - all chains converge in a finite number of steps.

[1]A flat domain isone in which $x \leq y$ implies $x = \perp$ or $x = y$

$[[f^{ub}]]$ *overestimates* $[[f]]^{ub}$, in that:

$\forall\, a \epsilon\, PD[A1], [[f^{ub}]]{:}a \supset [[f]]^{ub}{:}a$

As $[[f^{ub}]]$ overestimates the set of values an expression might produce, it is useful in situations where we wish to show that certain values (e.g. error values) *cannot* result from the evaluation of an expression.

We illustrate the utility of these ideas, with a number of practical applications. We motivate and solve the problems of inferring what we call *minor signatures* and *relevant clauses*. These have arisen out of our work in developing a strongly typed applicative language TFEL [14, 16] and are described in Subsections 1.2 and 1.3. In addition, we remark that the inference technique described in [18] can be described in our framework. In [18], abstract interpretation is used to optimize the implementation of *integer* valued applicative programs in a call-by-need programming language. A simple computation allows the inference of "strict" parameters of functions - i.e. those parameters that may be safely computed by call-by-value. This scheme has a simple description (a combination of join and meet interpretations) in our framework.

1.2. Minor signatures

Example 1.2.1 displays the type *integer list* together with functions *first* and *rest* defined on integer lists. The language we use is an extended applicative language, of the form called "equational" or "clausal", similiar to HOPE [3] or TFEL [14, 16]. Functions in such languages can be specified via their action on prototypical terms. All functions are strongly typed; types are specified by *data* equations. The symbol " + + " should be read as "or"; the symbol ":" stands for function application.

```
data list = nil + + fby[integer # list]

dec first: list -> integer
— first:fby[x,y] = x

dec rest: list -> list
— rest:fby[x,y] = y
```

Example 1.2.1

The data equation defines the type *list* of integers. All data are represented by *data constructors* applied to a number of subterms, each of which represents another data item. The type *list* has two constructors: *nil* and *fby*. *Fby* accepts terms of type *integer* and *list* and constructs a list out of them. Constructors form "records" of the appropriate type and arity, components of which are recoverable by (implied) *selector* functions. Functions *first* and *rest* are specified by *equations* or *clauses* describing their action on prototypical terms belonging to *list*.

The need for inferring *minor signatures* is motivated by noting that *first* and *rest* are *partial*[2] functions. Typically, an application of the form *first:nil* would generate an error at run-time. Fixing our terminology, let the *major signature* of a function be what is traditionally known as the signature (or type) of a function. In the case of *rest* the *major signature* is *list->list*. Given the *major signature* of a function, the *minor signature* characterizes the action of the function on terms belonging to the domain type of its major signature, indicating such things as:

1. *Is the function total ?* In the case of *rest*, given any term belonging to *list* (the domain type), does it yield a proper term belonging to *list* (the range type), or does it yield an error value ? Clearly *rest* is not total, as it yields an error value when applied to *nil*.
2. *If the function is partial, what are the terms belonging to its domain type for which it yields an error value ?* For *rest*, the only such term is *nil*.

More formally, we define:

Definition 1: Given function *f* with major signature *D -> D*, let *P* be a partition of *D*, then the minor signature of *f* with respect to *P* is a function from *P* to the powerset of *P* augmented with an error-value such that:

minor signature(f):a = $\{ b_i \}$
iff $\forall$ xϵa, [[f:x]]ϵb_i for some i

In sections that follow we will provide a precise description of possible partitions (via abstraction mappings), describe an appropriate power construction and carry out inference with respect to a particular partition.

In practice, we can only compute an *approximation* to the *minor signature*; this implies that we will certify only a *subset* of all total functions to be total. Instead of picking out the precise set of terms for which a function yields an error, we will pick out a superset of these terms.

The utility of performing such a minor signature analysis should be clear: information on the behaviour of functions is made available to the user at compile time; some erroneous expressions are detected before a program is run, and the run-time overhead of including error-handlers is reduced accordingly. Thus, inferring minor signatures is a form of compile-time detection of exceptions. In contrast, the language ML [7] explicitly incorporates such exceptions in the form of *failures*. Minor signatures are useful for static detection of errors, whereas *failures* in ML serve both as a means of indicating errors, *and* as an explicit programming technique.

[2]In this context, by partial function we mean a function that yields an error on being applied to a term in its defined domain. We are not referring to the possibility of non-termination.

1.3. Relevant clauses

The utility of inferring relevant clauses is illustrated by the following example. Consider the following definition of the type (univariate) *polynomial* and the function *value* which evaluates a polynomial at the integer *n*.

```
data poly =
      X + + const[integer]
       + + mul[poly # poly] + + add[poly # poly]
       + + sub[poly # poly] + + exp[poly # integer]

dec value: poly -> integer
[1]- value:[X,n]          = n
[2]- value:[const[p],n] = p
[3]- value:[mul[x,y],n] = value[x,n]*value[y,n]
[4]- value:[add[x,y],n] = value[x,n] + value[y,n]
[5]- value:[sub[x,y],n] = value[x,n] - value[y,n]
[6]- value:[exp[x,m],n] = value[x,n]^m
```

Example 1.3.2

The function *value* is specified by its action on every possible term belonging to type *poly*. In Example 1.3.2 we have attached a *clause number* to each clause. Frequently, only a few clauses will be needed to determine the result of a particular application of *value* — consider the expressions *value:[mul[X,const[3]],9]* and *value:[exp[X,7],9]*. In the first expression only the first three clauses, and in the second expression the first and last clauses, are of interest. We will call such clauses *relevant clauses.* To be precise, we should speak of *relevant clauses* of a function with respect to an application of the function to a term. Determining the relevant clauses of a function statically allows an interpreter to reduce the number of clauses it must inspect at run-time; this is of particular importance in *combinator* based (i.e. *copying*) interpreters [11] and systems based on tree rewriting [9], in which an application is effectively *replaced* with its definition. Such a definition will typically involve a run-time case analysis for minor signatures. By inferring minor signatures at compile time, the size of definitions which must be copied is substantially reduced.

More formally, we define:

Definition 2: Given function *f* with major signature *D -> D*, the *relevant clauses* of *f* with respect to a partition *P* of *D*, is a function from *P* to the powerset of clause numbers such that:

relevant clause(f):a = $\{k_i\}$
iff $\forall$ xϵa, clause k_i (for some i)
is relevant in the expression f:x.

Again, we will only be able to compute an *approximation* to the relevant clauses of a function, inferring a *superset* of the relevant clauses. Inferring relevant clauses of a function is related to the notion of *overloading* [1], in the sense that each clause defining a function can be considered to be an independent function definition (e.g. *value* above). In overloading, however, an *a priori* signature of an overloaded function is always available, thereby restricting the problem to one of choosing a single function from a set of functions. Inferring relevant clauses, on the other hand, involves both computing a "signature" for each clause of the function based on some description of terms (as described below) and, when given an application of the function to a term, choosing the *subset* of clauses which could be applicable.

1.4. Related Work

Abstract interpretation of imperative programs has been formalized by [5, 6]. In their framework abstract interpretation takes the form of modelling values of program variables at different points in a program. As we have earlier suggested, their construction involves power sets rather than power domains. Consequently, unlike the domains we consider, which are complete partial orders, the Cousots' work with complete lattices.

Mycroft [19] pioneered the extension of abstract interpretation to applicative programs. While he has not addressed the problems that arise in dealing with non–flat domains, we owe several crucial observations to him (see Section 3) and urge the interested reader to consult Chapter 2 of [19] for a comprehensive discussion of the abstract interpretation problem for applicative languages.

In the remainder of this work we describe a correctness result for the construction sketched in Figure I and apply this result to the problems discussed above. In section 2 we describe interpretations for *data* equations and function definitions, which form the basis for arriving at the *minor signature* and *relevant clause* of functions. Section 3 is an outline of the correctness result and provides the necessary machinery for abstracting the interpretations given in Section 2. In section 4 we describe solutions for the minor signature and relevant clause problems; section 5 describes an important example of the use of relevant clause inference.

2. Standard Interpretations

In this section we specify a denotational interpretation for *data* equations and provide *two* denotational interpretations for function definitions. The first relates function definitions to their standard denotations (i.e. namely abstract functions). The second interprets function definitions as functions from terms to sets of clause numbers. Building an approximation to the first definition yields the minor signature of a function; approximating the second definition yields the relevant clauses of a function. We also specify a "simplified" domain based on which we construct approximations.

We will use functions defined on the following types as examples:

data integer = zero + + succ[integer]

data bool = true + + false

Example 2.0.3

There are two popular styles of interpreting data equations as domains. The first, which we will call a "strict" domain, reduces to the insistence that a value is defined (non–bottom) only if all its sub–values are defined. This gives rise to a *flat* domain. "Lazy" domains result if we admit terms with components that may be undefined and lead to the possibility of "infinite objects" derived from cyclic constant definitions. Details of these constructions may be found in [12, 4]. As we are interested in non–flat domains, we use the second interpretation.

We capture error values by introducing constant *bad* (after [15]) into the domain of data values. As we are modelling non–strict functions, we cannot insist that functions be bad–preserving – yield *bad* on being applied to *bad*. Similarly "lazy" interpretations of data equations will generate terms with *bad* embedded in them. Figure II displays the structure of the domain of integer values, extended with *bad*, for both the "lazy" and "strict" interpretations. Note that we use type *integer* only to capture the essential structure of streams with minimal development of formal machinery. We are not suggesting that "lazy" implementations of type *integer* are useful in practice.

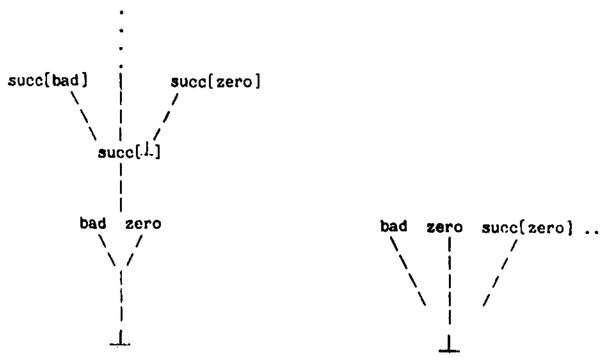

Figure II: Lazy and strict interpretations for integers

The two interpretations for expressions and functions definitions that we use are conventional and we do not describe them in any detail. The first, E_{fun}, maps expressions and function definitions to values and functions over the TOI (Type of Interest). Applying a function to an unexpected argument results in the value *bad*.

Example 2.0.4 demonstrates the action of E_{fun}, using function *sub1* defined on integers.

sub1:succ[zero] = zero
sub1:succ[succ[x]] = succ[sub1:succ[x]]

E_{fun}[[sub1:succ[zero]]] = zero
E_{fun}[[sub1:zero]] = bad

Example 2.0.4

Our second interpretation is more interesting. We number each clause in a function definition with a positive integer (1..n). The action of E_{cl} is to yield the clause numbers that are entered during execution. Operationally, this is equivalent to each clause in a function definition returning a pair, consisting of the computed value and the clause number. In the following discussion we ignore the computed value, but clearly it is essential in defining the interpretation. E_{cl} maps expressions into sets of clause numbers and interprets functions as mapping values (drawn from the TOI) to sets of clause numbers. Intuitively, E_{cl} *simulates* the evaluation of a function on being applied to an argument, collecting clause numbers as it does so. The result of applying a function to a value is a set of clause numbers describing the clauses entered during evaluation. The action E_{cl} on expressions can be thought of as yielding a collection of sets of clause numbers each labelled by the function name and function application from which the set is derived. In Example 2.0.5 the function *even*, (defined on integers), is mapped to a function from integers to sets of clause numbers.

[1]– even:zero = true
[2]– even:succ[zero] = false
[3]– even:succ[succ[x]] = even:x

E_{cl}[[even]] : integer –> PowerSet[Clause numbers]
= < <zero, { 1 } >, < succ[zero], {2} >,
<succ[succ[zero]], {1,3}>...>

E_{cl}[[even:succ[succ[succ[zero]]]]]
= [even, {3,2}]

E_{cl}[[even:sub1:succ[succ[zero]]]]
= [sub1, {1,2}][even, {2}]

Example 2.0.5

The particular domain (partition) which we use as an example is derived from the *set of the constructors* (written *Con*). For the type *integer*, augmented with *bad*, Con is derived from the set of constructors {*zero*, *succ*, $\bot$, bad}. It is straightforward to verify that the abstraction map Φ is *acceptable* in the terminology of section 3.

Φ: INTEGER ——> Con

Φ:$\bot$ = $\bot$
Φ:zero = zero
Φ:succ[x] = succ & Φ:x
Φ:bad = bad

succ & $\bot$ = succ

Definition of Φ

$$\begin{tikzcd}
\text{succ \& bad} & & \text{succ \& zero} \\
 & \text{succ} \arrow[ul, dash] \arrow[ur, dash] & \\
\text{bad} & & \text{zero} \\
 & \bot \arrow[ul, dash] \arrow[ur, dash] \arrow[uu, dash] &
\end{tikzcd}$$

Structure of *Con*

The operator "&" can be thought of as a form of infix union, with singleton subset *succ* & $\bot$ identified with *succ*. This ensures the continuity of Φ. Φ applied to an integer value reveals the constructors used to construct the value. Φ induces a map from *terms* and *constant equations* to *Con* in the obvious manner.

3. Correctness

In this section we provide an outline of the correctness proof for the techniques described in Section 1. Definitions of continuity, finite element, complete partial order (c.p.o), countably algebraic c.p.o (domain) and other domain theoretic notions used below can be found in [22].

Theorems 4 and 5 are standard in domain theory [21], and define a powerdomain construction for finite c.p.o.'s.

Definition 3: Let E be a c.p.o. A subset X of E is *convex* iff ($\forall$ x,y,zϵE. x $\leq$ y $\leq$ z and x,zϵX implies yϵX).
Let *Conv*(X) = { z | zϵE, x$\leq$z$\leq$y, x,yϵX }.

Theorem 4: [Plotkin] Given a finite c.p.o D, PD[D] = { *Conv*(X) | X $\subseteq$ D, X$\neq$$\emptyset$} , PD[D] is a c.p.o with $\leq$ defined to be: [Egli–Milner ordering]: a $\leq$ b iff $\forall$ yϵb there exists xϵa such that x $\leq$ y and $\forall$ xϵa there exists yϵb such that x $\leq$ y.

Theorem 5: For any c.p.o D, Closed Union (*Conv* $\cup$) is a continuous function on PD[D], and the subset relation ($\subseteq$) is continuous over PD[D].

We will also use the subset relation on functions over

PD[D]; this is a natural "lifting" to functions. Theorem 4 states that the union–based interpretation can be expressed in terms of the above powerdomain construction. In stating theorem 4, we need to place a technical restriction on the order structure of acceptable "simplified" domains and on the abstraction mapping. If abstraction mapping *Abs* maps domain *D* to *A*, we require the $\le$ order on *A* to to be related to the $\le$ order on *D*, in that $a_1, a_2 \epsilon A$ should be related only if elements of *D* from the pre–images under *Abs*, are related. The restriction on abstraction mapping states that the sets { x | Abs1:x = a_1 }, { x | Abs1:x = a_2 } must be related in a particular way.

Definition 6: Given domains *D* and *A*, continuous, total and onto abstraction mapping *Abs:D –> A*,is an an *acceptable abstraction mapping* iff for all a_1, $a_2 \epsilon A$, if $a_1 \le a_2$ then for all finite elements x∊{ r | Abs1:r = a_1} there exists y∊ { s | Abs1:s = a_2} with x≤ y, and for all finite elements y∊{ s | Abs1:s = a_2} there exists x∊{ r | Abs1:r = a_1} with x≤y.

In what follows we assume all abstraction mappings to be acceptable. This restriction on abstraction mappings is fairly complex, but we do not have a simpler characterization at this time.

Theorem 7: Given [[f]]:D –> R, D and R domains, acceptable abstraction maps *Abs1*: D –> A1, *Abs2*: R –> A2, A1 and A2 finite, domains, let:

$[[f]]^{ub}$:a = *Conv*[{Abs2:[[f]]:x |Abs1:x = a}]

then:

1. $[[f]]^{ub}$ is a continuous function from A1 –> PD[A2].
2. { Abs2:[[f]]:x | Abs1:x = a } ⊆ $[[f]]^{ub}$:a
3. If $[[f_1]]$, $[[f_2]]$:D –> R, $[[f_1]] \le [[f_2]]$ then $[[f_1]]^{ub} \le [[f_2]]^{ub}$.
4. $[[f]]^{ub}$ extends to $[[f]]^{UB}$:PD[A1] –> PD[A2], $[[f]]^{UB}$:X = Conv { $[[f]]^{ub}$:a | a∊X }.

Proof: We outline (1) above, parts (2), (3) and (4) are straightforward. For $a_1 \le a_2$ we need to show $[[f]]^{ub}:a_1 \le [[f]]^{ub}:a_2$.
From the definition of acceptable abstraction mapping we have, for all finite elements s∊{ x | Abs1:x = a_1 } there exists r∊{ y | Abs1:y = a_2} with s ≤ r and vice–versa.
Further Abs2°[[f]] is a continuous function from D to A2. Then { Abs2:[[f]]:x | Abs1:x = a_1 } ≤ { Abs2:[[f]]:x | Abs1:x = a_2}, as A2 is finite.

Each f:D –> R induces a map from A1 to PD[A2] and can further be embedded into PD[A1] –> PD[A2]. Note that securing composition of functions is the only reason to consider PD[A1] –> PD[A2]; indeed it can be verified that the embedding yields a closed sub–space of PD[A1] –> PD[A2] consisting of the "natural extension" of the function space A1 –> PD[A2]. We freely identify these interpretations in future development.

In 7.2, for *flat* domains we have the relationship $[[f]]^{ub}$:a = { Abs2:f:x | Abs1:x = a }. For non–flat domains, the relationship is weakened to an inclusion. The well known "convex hull" problem [2], caused by identifying sets with their convex closures, forces us to make the weaker statement on non–flat domains. In pragmatic terms this implies some extra loss of information during inference.

The union–based interpretation is useful when the set of all possible results is of interest; two subsidiary interpretations, the *meet* and *join* interpretations, yield upper and lower bounds of the set of results, and are often simpler to use in applications. The *meet* (or *join*) interpretations are only available when the range (A2, in A1 –> A2) is a meet– (or join)–complete (closed under greatest lower bounds (least upper bounds) of subsets of A2).

Theorem 8: Given [[f]]:D –> R, D and R domains, acceptable abstraction maps *Abs1*: D –> A1, *Abs2*: R –> A2, A1 and A2 finite domains, let:

$[[f]]^{j}$:a = $\bigsqcup$[{Abs2:[[f]]:x | Abs1:x = a}]

$[[f]]^{m}$:a = $\bigsqcap$[{Abs2:[[f]]:x | Abs1:x = a}]

then:

1. $[[f]]^{j}$,$[[f]]^{m}$ are continuous functions from A1 –> A2.
2. ∀z∊{ x | Abs1:x = a }, Abs2:[[f]]:z ≥ $[[f]]^{m}$:a
3. ∀z∊{ x | Abs1:x = a }, Abs2:[[f]]:z ≤ $[[f]]^{j}$:a
4. If $[[f_1]]$, $[[f_2]]$:D –> R, $[[f_1]] \le [[f_2]]$ then $[[f_1]]^{m}$ ($[[f_2]]^{j}$) ≤ $[[f_2]]^{m}$ ($[[f_2]]^{j}$).

Proof: Straightforward from Theorem 7 and continuity of meet and join over the powerdomain (p.477 [21]).

Below we present an application of the union–based and join abstract interpretations for the minor signature and relevant clause problems:

Union–based, D = integer, R = integer,
Abs1 = Abs2 = Φ, A1 = A2 = Con

E_{fun}[[sub1]]^{ub} =
zero –> {bad}, succ & zero –> {zero, succ & zero},
⊥ –> {⊥}, succ –> {succ}
bad –> {bad}, succ & bad –> {succ & bad,bad}

Example 3.0.6: Minor signature of *sub1*

For the relevant clause problem, the availability of a complete lattice (A2 = Powerset[Clause Numbers]) with ≤ equal to ⊆ suggests that the join interpretation should suffice:

Join, D = integer, R = PowerSet[Clause Numbers],
Abs1 = Φ, Abs2 = identity,
A1 = Con, A2 = PowerSet[Clause Numbers]

E_{cl}[[even]]^j =

zero	-> {1},	succ & zero	-> {1,2,3}
$\perp$	-> {},	succ	-> {3}
bad	-> {},	succ & bad	-> {3}

Example 3.0.7: Relevant clauses of *even*

Theorems 7 and 8 are not helpful in actually computing $[[f]]^{ub/j/m}$, as $[[f]]^{ub/j/m}$ are defined via the standard denotation. We need to derive $[[f^{ub/j/m}]]$ as approximations to $[[f]]^{ub/j/m}$ from the function representation for f.Theorems 7 and 8 are applied "piecewise" to primitive interpreted functions (if–then–else etc.) inducing an interpretation for function defintions via composition and by taking fix–points. Before doing so we need some results on function application and composition:

Theorem 9:

1. $([[f]]^{ub} \circ [[g]]^{ub}) \supseteq ([[f]] \circ [[g]])^{ub}$,
2. $([[f]] \circ [[g]])^{j} \le [[f]]^{j} \circ [[g]]^{j}$,
3. $([[f]] \circ [[g]])^{m} \ge [[f]]^{m} \circ [[g]]^{m}$

Proof: For (1) above, notice that $[[f]]^{ub}:[[a]]^{ub} \supseteq [[f:a]]^{ub}$. For (2) and (3) similarly.

We write recursive function definitions as f = E[f] interpreted as usual by the taking of fix–points with respect to a set of primitive functions symbols { c_i }. For the union–based interpretation we have:

Theorem 10: Given a function representation f = E[f] for function [[f]]:D -> R, Abstraction maps Abs1:D -> A1, Abs2: R -> A2:

$[[f]]^{ub} \subseteq [[f^{ub}]]$

where $[[f^{ub}]] = \lim_i (E^{ub})^i [\perp]$

where $E^{ub}[f] = E[f] <c'_i/c_i> \forall i$,

and $[[c'_i]] = [[c_i]]^{ub}$.

Proof: From $[[c_i]] \subseteq [[c'_i]]$ and Theorem 9 we have the relationship for finite compositions of primitive functions, and we only need to show that the relationship holds for limits. From $E^n[\perp] \subseteq (E^{ub})^n[\perp]$, and the continuity of the subset predicate we have the required relationship.

The role of two partial orders ($\subseteq$, $\le$) over the underlying powerdomain for the union–based interpretation has been remarked upon by [19]. The $\le$ ordering captures improvement during the process of iterating to the fixed point; the $\subseteq$ ordering captures loss of information due to coarseness of the abstract interpretation.

The correctness result for $[[f]]^j$ and $[[f]]^m$ follows by an argument similar to that used above:

Theorem 11: $[[f^{jm}]] \le [[f]]^m \le [[f]] \le [[f]]^j \le [[f^j]]$

Proof: As above, using continuity of meet and join as relations.

The *intersection–based* interpretation (the dual of the union–based interpretation) defined by $[[f]]^{ib}:a = \cap$ { {*Abs2:[[f]]:x*} | *Abs1:x = a* } is *not* available in general, unless the underlying powerdomain has a natural intersection operation. The intersection–based interpretation is useful when we need to show that a certain value *must* result from the evaluation of an expression. It is not clear whether it is possible to consistently extend the underlying domain to permit such an operation.

Finally, we compare our constructions with those of [5] and [19]. We have earlier in Section 1 pointed out some specific differences. We further note that our construction is developed in a non–standard fashion and limited to a restricted set of abstraction maps. Traditionally abstract interpretations of a standard interpretation are formulated as abstractions of the "collecting interpretation" – the natural lifting of the standard semantics to the powerdomain (or powerset) of the underlying domain (or set). Our development of abstract interpretation is therefore fairly restrictive. The traditional setting has the advantage that interpretations can easily be compared (see Cousots' lattice of abstract interpretations) and more complex abstraction maps can be expressed.

4. Solving for Minor Signatures and Relevant clauses

We discuss some pragmatic details of the minor signature and relevant clause inference system. The system carries out both inferences in sequence; minor signatures are inferred first and used to drive the relevant clause inference system. Function definitions are first mapped into an appropriate form for solution by computation of least fixed points. For minor signature inference, this implies non–recursive function definitions take on functionality *PD[Con] -> PD[Con]*, with recursive definitions appearing as functionals over the same space. For relevant clause inference the functionality is *Con -> PowerSet[Clause Numbers]*.

The transformation is straightforward, save for left hand sides of clauses (called *pattern predicates*) which require some pre–processing. As pointed out in Section 3 primitive functions are simply re–interpreted over the abstract domains. This works well for functions such as if–then–else, but before re–interpreting clauses it is necessary to carry out some extra pre–processing. Define the *intersection set* of a pattern predicate to be those elements of *Con* having a pre–image in the TOI containing elements which might match the pattern predicate. For *x* in *f:x = e_1* the intersection set is all of *Con*; for *Succ[x]* in *f:succ[x] = e_1* the intersection set is {succ & zero, succ, succ & bad}. Using intersection sets we produce clause definitions over *Con*, as shown below in Example 4.8 for *sub1*:

$$
\begin{array}{ll}
\text{sub1:succ \& zero} & = \text{zero} \cup \text{succ \& sub1:(succ \& zero)} \\
\text{sub1:succ \& bad} & = \text{succ \& sub1:(succ \& bad)} \cup \text{bad} \\
\text{sub1:}\bot & = \bot \\
\text{sub1:bad} & = \text{bad} \\
\text{sub1:zero} & = \text{bad} \\
\text{sub1:succ} & = \text{succ \& sub1:succ} \\
 & \ \cup \text{succ \& sub1:}\bot
\end{array}
$$

Example 4.8: Transformed version of Sub1

Memebers of *Con*, that do not belong belong to any intersection set show up mapping to *bad* (except for $\bot$) in the re–interpreted clauses. Currently, the re–interpreted clauses are solved for using a simple iterative algorithm. Example 4.9 displays the inferred *minor signature* for *sub1*, computed from the representation in Example 4.8:

$$
\begin{array}{l}
E_{fun}[[\text{sub1}^{ub}]] = \\
\text{bad} \ \text{->} \{\text{bad}\}, \ \text{succ} \text{->} \{\text{succ}\}, \\
\text{zero} \ \text{->} \{\text{bad}\}, \ \bot \ \text{->} \{\bot\}, \\
\text{succ \& zero} \quad \text{->} \{\text{succ \& zero,zero,succ}\}, \\
\text{succ \& bad} \quad \text{->} \{\text{succ \& bad, bad, succ}\},
\end{array}
$$

Example 4.0.9: $E_{fun}[[\text{sub1}^{ub}]]$

The over–estimate of results inferred by $E_{fun}[[\text{sub1}^{ub}]]$ is visible when compared with $E_{fun}[[\text{ sub1 }]]^{ub}$ in Section 3. For the value *succ & zero*, $E_{fun}[[\text{sub1}^{ub}]]$ yields { succ & zero, zero, succ}, as opposed to {succ & zero, zero} suggested by $E_{fun}[[\text{ sub1 }]]^{ub}$. Basically the inference algorithm fails to infer termination and consequently throws in the extra value *succ*.

Restating the goals of minor signature inference discussed in section 1 in terms of *Sub1*: *Sub1* is a partial function, yielding an error when applied to *zero*. Error–handlers need only be included when the minor signature suggests that *bad* occurs amongst the set of possible results. From $E_{fun}[[\text{ f }]]^{ub} \subseteq E_{fun}[[f^{ub}]]$ we have:

- If $E_{fun}[[\text{ } f^{ub}]]$:a = { bad } for some a$\in$*Con*, then for all values *x* in the pre–image of a, E_{fun} [[f:x]] = *bad*.
- If for some *x*, E_{fun}[[f:x]] = bad, then bad$\in E_{fun}$ [[f^{ub}]]:Abs1:x The presence of *bad* in the set of results can be used as a *necessary* condition for the inclusion of an error handler.
- Finally, if *bad* does not occur in a set of results, i.e. *bad* is absent from $E_{fun}[[f^{ub}]]$:a for some a$\in$*Con*, then no error can occur for values drawn from the pre–image of a, i.e E_{fun}[[f:x]] <> *bad* where *x* such that Abs1:x = a.

Computing relevant clauses follows the general outline suggested above. As we are using the join interpretation, we have $E_{cl}[[\text{ f }]]^{j} \le E_{cl}[[\text{ } f^{j} \text{ }]]$, and will in general infer a superset of the relevant clauses. The meet interpretation could also be used to infer a subset (all clauses that *must* be utilized for a particular application), but is not of practical interest.

Example 4.10 displays the inferred *relevant clauses* for *even*. Note that we cannot solve the relevant–clause problem without inferring the minor signature of functions concerned, as at all times we require estimates of the values being passed to functions and the values returned by them:

$$
\begin{array}{l}
E_{cl}[[\text{even}^{ub}]] = \\
\text{zero} \ \text{->} \{1\}, \ \text{succ \& zero} \text{->} \{1,2,3\}, \\
\bot \quad \text{->} \emptyset, \ \text{succ} \qquad\quad \text{->} \{3\} \\
\text{bad} \ \text{->} \emptyset, \ \text{succ \& bad} \ \text{->} \{3\}
\end{array}
$$

Example 4.10: $E_{cl}[[f^{j}]]$

Comparing with $E_{cl}[[\text{ even }]]^{j}$ in section 3, we find $E_{cl}[[\text{ even}^{j} \text{ }]]$ to be identical; this will not be the case in general.

Finally, we note some points where the actual inference system differs from the theoretical basis. As noted in Section 3, using the union–based interpretation forces the identification of subsets of *Con* with their convex closures. This forces the identification of {$\bot$, succ & zero} and {$\bot$, succ, succ & zero}. We are currently investigating whether we can avoid this identification both for non–recursive functions (straightforward) and (use a looping test) for recursive definitions.It remains to be seen if this is well–founded.

5. Relevant clause inference in a language intended for concurrent execution

We describe the important role played by relevant clause inference in optimizing the execution of the functional language FEL [14] on a reduction–based multiprocessor [11]. In [13] a method was given whereby sequences could be stored either as Lisp *dotted–pairs*, as contiguous blocks (called *tuples*), or as contiguous blocks representing virtual concatenations of sequences (called *concs*). In other words, the underlying representation of a given sequence might use these constructs in any combination and to any number of levels. An obvious reason to prefer one representation over another is the accessibility/modifiability trade–off: Access of the *ith* component of a dotted–pair representation requires time linear in i, whereas access of a tuple requires constant time; Access of a virtual concatenation is somewhere in between, depending on *balance*. On the other hand, since the possibility of shared or concurrent access to sub–structures preculdes in–place modification (a new structure must be created), pure tuples are the most expensive to (virtually) modify.

Subsequently, a set of sequence operators was provided in FEL for performing commonly-used operations on these generic sequences. The intention here was to relieve the programmer from having to think about several different versions of operators which do similar operations, and to provide optimized implementations of these operators which could exploit the potential for concurrent execution in an applicative multiprocessor. For example, the generalization of Lisp's *mapcar* written as || (called *parallel application*) is such that f || x applies a function f component-wise to *any* sequence x.Similarly, if fs is a sequence of functions, then fs :: x applies each f in the sequence to x, and so on. In implementing such generic operators, the general case demands the inclusion of a run-time test for the representation type of the sequence at each level of recursion. However, in many applications, only one of the representations is actually used.

Consider the function *rest* as it might be defined on integer sequences with the generic representation described above. Example 5.11 illustrates the clauses in a format similar to that used in TFEL:

```
data intlist
⩦ nil + + list[integer tuple]
   + + fby[integer,intlist]
   + + conc[intlist tuple]]

dec rest : intlist -> intlist
— rest:list[x]   = list[t—rest:x]]
— rest:fby[x,y] = y
— rest:conc[x] = if t—first:x = nil then
                 rest:conc[t—rest:x]]
                 else conc[rest:t—first:x,t—rest:x]
```

Example 5.11: integer lists

The function *t—first* yields the first element of a tuple; function *t—rest* yields the tuple derived by removing the first element. The constructors *nil* and *fby* are conventional; the constructor *list* builds a list from an arbitrary number of integers and is stored as a tuple, the constructor *conc* (standing for concatenation) also builds a tuple of its arguments. The extractor functions, (e.g. *rest* above) interpret *conc* and *list* appropriately, effectively causing them to possess the same functional semantics as *append* or *list* would in Lisp.

Use of *conc* and *list* therefore yields lists that are (often) structured as blocks rather than pairs, achieving the objectives described above. However, this generous use of constructors forces supporting functions (e.g. *rest* above) to be be written with many clauses. In a copying reduction-based system this implies excessive memory utilization (through function body copying).

The relevant clause and minor signature inference scheme described herein permits us to optimize the implementation by excluding tests for representations which have been inferred not to be relevant. This scheme is thus a powerful device for freeing the programmer from detailed concerns about representations, permitting easy change from one to another, and allowing greater experimentation to find which representation yields the best possibilities for concurrent execution. An implementation in full FEL, incorporating both minor signature inference and relevant clause inference, is underway.

6. Conclusion

We have argued that abstract interpretation provides an useful framework in which to develop inference schemes for applicative languages. We have presented several practical examples of its utility and outlined a simple correctness proof for the abstract interpretations we use.

We are currently extending the work in several directions. The correctness proof contained in this paper is fairly ad-hoc (restriction to finite domains etc.) and we are in the process of extending it [20]. We are also examining several other inference problems that include extending [18] to list structures in general; re-phrasing, and thereby extending to functions, the cycle-sum test in [23] as well as describing the aggregate update problem [10] as an inference problem.

A prototype implementation that infers relevant clauses and minor signatures is operational. An implementation in full FEL is underway. Details of algorithms used and gains due to optimization will appear in [17].

Acknowledgements

We thank Prakash Panangaden, Uday Reddy and Esther Shilcrat for helpful discussions and in aiding the preparation of this report.

References

1. United States Department of Defense. *Reference manual for the ADA programming language.* United states Department of Defense, 1982.

2. M. Broy. Fixed point theory for communication and concurrency. Lectures at the International Summer school on Theoretical Foundations of Programming Methodology, July, 1981, pp. .

3. Burstall R.M. , MacQueen D.B. and Sanella D.T. HOPE: An Experimental Applicative Language. 1980 LISP Conference, 1980, pp. 136–143.

4. Cartwright R., Donahue J. The semantics of Lazy (And Industrious) Evaluation. Symposium on Functional Languages and LISP, August, 1982, pp. 253–264.

5. P. Cousot and R.Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." *POPL IV* (Jan 1977), 238–252.

6. P. Cousot. Semantic Foundations of program analysis. In N.Jones and N.Muchnick, Ed., *Program Flow Analysis: Theory and Applications*, Prentice–Hall, 1981, pp. 303–342.

7. M. Gordon, R.Milner, L.Morris, M.Newey, C.Wadsworth. A Metalanguage for Interactive Proof in LCF. Fifth POPL, 1978, pp. 119–130.

8. P. Henderson. *Functional programming.* Prentice–Hall, 1980.

9. Hoffman C.M., O'Donnell J. "Programming with Equations." *TOPLAS 4*, 1 (Jan. 1982), 83–111.

10. P. Hudak. The aggregate update problem in functional programming systems. In preparation, Yale University, 1983

11. R.M. Keller, G.Lindstrom, and S.Patil. A loosely–coupled applicative multi–processing system. AFIPS, AFIPS, June, 1979, pp. 613–622.

12. R. M.Keller. Semantics and Applications of Function Graphs. Tech. Rept. UUCS–80–112, University of Utah, Computer Science Department, 1980.

13. R.M. Keller. Divide and CONCer: Data structuring for applicative multiprocessing. Proc. 1980 Lisp Conference, August, 1980, pp. 196–202.

14. R. M.Keller. FEL Manual. University of Utah, 1983.

15. R. Kieburtz. Precise typing of data type specifications. POPL X, Jan., 1983, pp. 109–116.

16. P. Mishra. Data Types in Applicative Languages: Abstraction and Inference. Ph. D.proposal, May 1983

17. P. Mishra, R.M. Keller. Optimized execution of a strongly typed applicative language. To appear, University of Utah,1983

18. A. Mycroft. The theory and practice of transforming call–by–need into call–by–value. In *LNCS 83*, LNCS 83, Springer–Verlag, 1980, pp. 269–281.

19. A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. Th., University of Edinburgh, December 1981.

20. P. Panangaden, P.Mishra. General powerdomain constructions for abstract interpretation and indeterminacy. In preparation, University of Utah, 1983

21. G. D.Plotkin. "A Powerdomain Construction." *SIAM J.Comput. 5*, 3 (Sept. 1976), 452–480.

22. M. B.Smyth. "Power Domains." *JCSS 16* (1978), 23–36.

23. W. Wadge. An Extensional treatment of Dataflow Deadlock. In G.Kahn, Ed., *Semantics of Concurrent Computation*, Springer–Verlag, 1979, pp. 285–299.