Introduction to Abstract Interpretation

1995

Mads Rosendahl
DIKU, Computer Science
University of Copenhagen

Abstract interpretation is a tool for constructing semantics based program analyses. These notes are written for the Introduction to Semantics course and assume knowledge of the Introduction to Domain Theory notes. They present some of the basic ideas in abstract interpretation using examples of program analyses expressed in this framework.

The current version is still not completely finished. Suggestions for improvements and corrections are most welcome.

1 Abstract interpretation

Abstract interpretation is a semantics-based program analysis method. The semantics of a programming language can be specified as a mapping of programs to mathematical objects that describes the input-output function for the program. In an abstract interpretation the program is given a mathematical meaning in the same way as with a normal semantics. This however is not necessarily the standard meaning, but it can be used to extract information about the computational behaviour of the program.

1.1 Abstract interpretation

The central idea in abstract interpretation is to construct two different meanings of a programming language where the first gives the usual meaning of programs in the language, and the other can be used to answer certain questions about the runtime behaviour of programs in the language.

The standard meaning of programs can typically be described by their input-output function, and the standard interpretation will then be a function $\mathbf{I}_1$ which maps programs to their input-output functions.

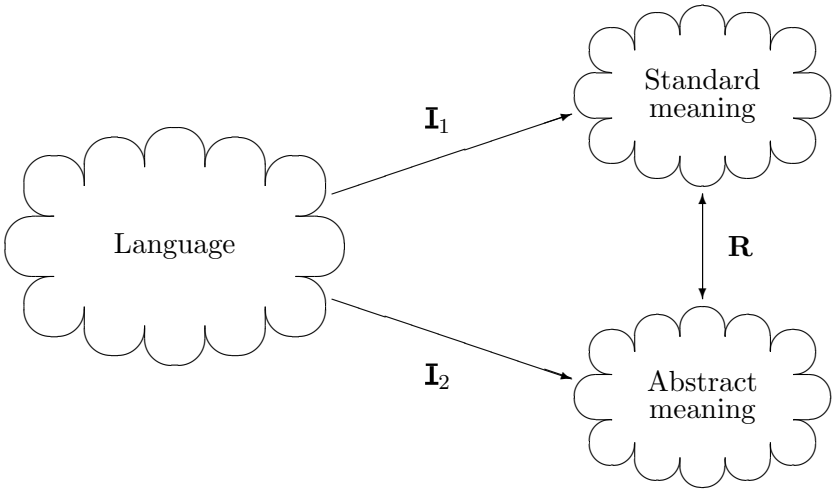
The abstract meaning will be defined by a function $\mathbf{I}_2$ which maps programs to mathematical objects that can be used to answer the question raised by a program analysis problem.

The correctness (or soundness) of this approach to program analysis can be established by proving a relationship between these two interpretations. An abstract interpretation of a language consists of the two function $\mathbf{I}_1$ and $\mathbf{I}_2$ together with a relation $\mathbf{R}$ between the meanings provided by these functions such that for all programs $\mathbf{p}$ the relationship holds:

$$\mathbf{I}_1[\![\mathbf{p}]\!] \ \mathbf{R} \ \mathbf{I}_2[\![\mathbf{p}]\!]$$

The relationship $\mathbf{R}$ between the two meanings describes which real program behaviours are described by an abstract meaning.

This can be sketched as



A final, fourth, component of an abstract interpretation is an intended use or application of the abstract meaning. This last part is normally the least formal. The abstract information is often used to guide a transformation or implementation of the program being analysed, and it is then argued that the resulting program behaves in an equivalent way to the original.

The aim in abstract interpretation is twofold. We prove the correctness of the abstract semantics with respect to the standard semantics and then use the specification of the semantics as a basis for an implementation. Compared to more *ad hoc* methods of program analysis such a framework may add a high degree of reliability and a structure which facilitates the specification of more complex analyses.

This general scheme has been used in a number of variations. The two meanings of a program are normally expressed in the form of fixed point semantics and the relationship can be constructed from functions which map abstract meanings to sets of possible standard meanings.

1.2 Overview

The notes are centered around some examples of abstract interpretations.

- Strictness Analysis of lazy functional languages.
- Live Variable Analysis of an imperative language.
- Groundness Analysis of a logical language. (This part is not yet included in the notes).

- Implementation techniques for abstract interpretation.
- Top-down vs. bottom-up analysis.

1.3 Rule-of-sign

The general idea of abstract interpretation is to interpret program text using some kind of non-standard values. The standard introductory example is the familiar *rule-of-sign* interpretation of integer expressions. We may in some situations be able to predict whether the result of an expression is positive or negative by only using the sign of the constants in the expression. Let us consider an expression built from integer constants, addition and multiplication. In the real world the expression can be evaluated as one may expect giving an integer result.

As an example consider the expression $-413 * (2571 + 879)$. We can deduce that the result is negative without actually performing the addition and the multiplication since we know that adding two positive numbers gives a positive result and that multiplying a negative and a positive number gives a negative result. We will here make this a bit more formal while introducing the notation used in the rest of these notes.

Syntax. Expressions are built from constants, addition and multiplication using this little grammar.

$$
\begin{array}{lll}
\mathbf{exp} ::= \mathbf{n} & & \text{number} \\
\quad | \quad \mathbf{exp} + \mathbf{exp} & & \text{addition} \\
\quad | \quad \mathbf{exp} * \mathbf{exp} & & \text{multiplication}
\end{array}
$$

Standard interpretation. The usual evaluation or interpretation of expressions can be specified by a function which computes the value of the expression. This evaluation function will here be written as a semantic function from denotational semantics. The shape or form of brackets and function symbols, however, is not important.

$$
\begin{array}{l}
\mathsf{E}_{\mathbf{std}}[\![\mathbf{exp}]\!] : \mathbb{Z} \\
\mathsf{E}_{\mathbf{std}}[\![\mathbf{n_i}]\!] = \mathbf{n_i} \\
\mathsf{E}_{\mathbf{std}}[\![\mathbf{exp}_1 + \mathbf{exp}_2]\!] = \mathsf{E}_{\mathbf{std}}[\![\mathbf{exp}_1]\!] + \mathsf{E}_{\mathbf{std}}[\![\mathbf{exp}_2]\!] \\
\mathsf{E}_{\mathbf{std}}[\![\mathbf{exp}_1 * \mathbf{exp}_2]\!] = \mathsf{E}_{\mathbf{std}}[\![\mathbf{exp}_1]\!] * \mathsf{E}_{\mathbf{std}}[\![\mathbf{exp}_2]\!]
\end{array}
$$

Abstract interpretation. It is not always possible to predict the sign of an expression from the sign of constants. The result of adding a positive and a negative number can both be positive and negative. In our sign interpretation we will operate with four different values: {zero, pos, neg, num} where zero indicates that the number is zero, pos that the number is positive, neg that the number is negative, and num that we don't know. We will call the set of these values **Sign**.

$$\mathbf{Sign} = \{\mathsf{zero}, \mathsf{pos}, \mathsf{neg}, \mathsf{num}\}$$

The rule-of-sign interpretation of addition and multiplication can then be specified in two tables. These interpretations may be viewed as binary operations on signs or as "abstract" additions and multiplications. We will write them as infix operations: $\oplus$ and $\otimes$ respectively.

$$\oplus : \mathbf{Sign} \times \mathbf{Sign} \to \mathbf{Sign} \qquad \otimes : \mathbf{Sign} \times \mathbf{Sign} \to \mathbf{Sign}$$

$\oplus$	zero	pos	neg	num
zero	zero	pos	neg	num
pos	pos	pos	num	num
neg	neg	num	neg	num
num	num	num	num	num

$\otimes$	zero	pos	neg	num
zero	zero	zero	zero	zero
pos	zero	pos	neg	num
neg	zero	neg	pos	num
num	zero	num	num	num

Recalling the example above, the abstract version of $-413 * (2571 + 879)$ is $\mathsf{neg} \otimes (\mathsf{pos} \oplus \mathsf{pos})$ which can be evaluated as neg.

Using these binary operations we can now define the abstract evaluation function for expressions. This function will compute the "sign" of the result.

$$\begin{aligned}
&\mathbf{E_{ros}}[\![\mathbf{exp}]\!] : \mathbf{Sign} \\
&\mathbf{E_{ros}}[\![\mathbf{n_i}]\!] &&= \mathbf{sign}(\mathbf{n_i}) \\
&\mathbf{E_{ros}}[\![\mathbf{exp}_1 + \mathbf{exp}_2]\!] &&= \mathbf{E_{ros}}[\![\mathbf{exp}_1]\!] \oplus \mathbf{E_{ros}}[\![\mathbf{exp}_2]\!] \\
&\mathbf{E_{ros}}[\![\mathbf{exp}_1 * \mathbf{exp}_2]\!] &&= \mathbf{E_{ros}}[\![\mathbf{exp}_1]\!] \otimes \mathbf{E_{ros}}[\![\mathbf{exp}_2]\!]
\end{aligned}$$

where

$$\mathbf{sign}(\mathbf{x}) = \mathbf{if}\ \mathbf{x} > 0\ \mathbf{then}\ \mathsf{pos}\ \mathbf{else\ if}\ \mathbf{x} < 0\ \mathbf{then}\ \mathsf{neg}\ \mathbf{else}\ \mathsf{zero}$$

Relation. The standard and the "sign" evaluation functions are closely related. They work on different kinds of values but the flow of information is the same. We may formalise the connection between signs and numbers by considering signs as representing the set of numbers with the given sign. In this way zero represents the singleton set $\{0\}$, pos represents the set of all positive numbers, neg represents the set of negative numbers and num is any number in $\mathbb{Z}$. Thus signs represent certain sets of numbers and we may also, given a non-empty set of numbers, find a description of its sign. For a given set we let its sign be the sign that denotes the least set containing the given set.

$$\begin{array}{ll}
\gamma : \mathbf{Sign} \to \mathcal{P}(\mathbb{Z}) \setminus \{\varnothing\} & \alpha : \mathcal{P}(\mathbb{Z}) \setminus \{\varnothing\} \to \mathbf{Sign} \\
\gamma(\mathsf{zero}) = \{0\} & \alpha(\mathbf{X}) = \textbf{if } \mathbf{X} = \{0\} \textbf{ then } \mathsf{zero} \textbf{ else} \\
\gamma(\mathsf{pos}) = \{\mathbf{x} \mid \mathbf{x} > 0\} & \qquad \textbf{if } \forall \mathbf{x} \in \mathbf{X}.\ \mathbf{x} > 0 \textbf{ then } \mathsf{pos} \textbf{ else} \\
\gamma(\mathsf{neg}) = \{\mathbf{x} \mid \mathbf{x} < 0\} & \qquad \textbf{if } \forall \mathbf{x} \in \mathbf{X}.\ \mathbf{x} < 0 \textbf{ then } \mathsf{neg} \textbf{ else } \mathsf{num} \\
\gamma(\mathsf{num}) = \mathbb{Z} &
\end{array}$$

The relationship between γ and α is then that

$$\begin{array}{l}
\forall \mathbf{s} \in \mathbf{Sign}.\ \ \alpha(\gamma(\mathbf{s})) = \mathbf{s} \\
\forall \mathbf{X} \in \mathcal{P}(\mathbb{Z}) \setminus \varnothing.\ \ \gamma(\alpha(\mathbf{X})) \supseteq \mathbf{X}
\end{array}$$

These functions are called *concretisation* ($\gamma \simeq \mathbf{c}$) and *abstraction* ($\alpha \simeq \mathbf{a}$) functions.

Using these functions we may explain how addition and multiplication were defined on signs

$$\begin{array}{l}
\mathbf{s}_1 \oplus \mathbf{s}_2 = \alpha(\{\mathbf{x}_1 + \mathbf{x}_2 \mid \mathbf{x}_1 \in \gamma(\mathbf{s}_1) \wedge \mathbf{x}_2 \in \gamma(\mathbf{s}_2)\}) \\
\mathbf{s}_1 \otimes \mathbf{s}_2 = \alpha(\{\mathbf{x}_1 * \mathbf{x}_2 \mid \mathbf{x}_1 \in \gamma(\mathbf{s}_1) \wedge \mathbf{x}_2 \in \gamma(\mathbf{s}_2)\})
\end{array}$$

Safety. The relationship between the standard and sign evaluation function may be stated as the following relation.

$$\forall \mathbf{exp}.\ \{\mathsf{E}_{\mathbf{std}}[\![\mathbf{exp}]\!]\} \subseteq \gamma(\mathsf{E}_{\mathbf{ros}}[\![\mathbf{exp}]\!])$$

Notice, however, that $\alpha(\{\mathsf{E}_{\mathbf{std}}[\![\mathbf{exp}]\!]\})$ and $\mathsf{E}_{\mathbf{ros}}[\![\mathbf{exp}]\!]$ not necessarily are equal. In other words, the sign evaluation function may not be able to predict the sign of the expression but if it does it will be correct. The relationship is often referred to as a *safety* or *soundness* condition. Safety proofs normally contain two parts: a local proof based on structural induction and a global proof based on fixpoint induction. In this case we only need the first part since no fixpoints are involved in the evaluation.

Structural induction over expressions is similar to normal induction. The induction start is to prove the given property for simple expressions:

$$\forall \mathbf{n} \in \mathbb{Z}.\ \{\mathsf{E}_{\mathbf{std}}[\![\mathbf{n}]\!]\} \subseteq \gamma(\mathsf{E}_{\mathbf{ros}}[\![\mathbf{n}]\!])$$
or $$\forall \mathbf{n} \in \mathbb{Z}.\ \{\mathbf{n}\} \subseteq \gamma(\mathbf{sign}(\mathbf{n}))$$

This follows by case analysis on the possible signs of numbers.

In the induction step we assume the property has been proved for subexpressions and prove that it holds for composite expressions.

$$\forall \mathbf{n}_1, \mathbf{n}_2 \in \mathbb{Z}, \mathbf{s}_1, \mathbf{s}_2 \in \mathbf{Sign}.\ \{\mathbf{n}_1\} \subseteq \gamma(\mathbf{s}_1) \wedge \{\mathbf{n}_2\} \subseteq \gamma(\mathbf{s}_2) \Rightarrow$$
$$\{\mathbf{n}_1 + \mathbf{n}_2\} \subseteq \gamma(\mathbf{s}_1 \oplus \mathbf{s}_2) \quad \wedge \quad \{\mathbf{n}_1 * \mathbf{n}_2\} \subseteq \gamma(\mathbf{s}_1 \otimes \mathbf{s}_2)$$

This follows from the relations between α, γ and the operations on signs.

From the induction start and the induction step we conclude that the relation holds for all finite expressions built from numbers using addition and multiplication.

2 Strictness analysis

A function $\mathbf{f}$ on a domain $\mathbf{D}$ is said to be *strict* if it maps the bottom element $\perp$ to the bottom element. In symbols it means that

$$\mathbf{f}(\perp) = \perp$$

Why is this so interesting? Well, this rather trivial property has been the foundation for one of the most important semantics based program analyses. If $\mathbf{f}$ is a function in a lazy functional language with this property it means that if the computation of the argument does not terminate then the computation of the function result will not terminate either. Hence if we compute the argument before the call we then have two possibilities: either the argument can be evaluated and no harm has been done or the evaluation of the argument will fail to terminate. In the latter case, with a strict function, we know that the computation of the function result would have failed anyway. The only difference is that it now may happen a bit earlier. Strictness of a function means that we may use a *call-by-value* strategy rather than *call-by-need* and this should hopefully make it possible to implement the function more efficiently.

There can be two reasons why a function of one variable can be strict. The function will either use its argument in all circumstances as in say,

$$\mathbf{f}(\mathbf{x}) = \textbf{if } \mathbf{x} = 0 \textbf{ then } 1 \textbf{ else } \mathbf{x} * \mathbf{f}(\mathbf{x} - 1)$$

or the function will not terminate for any argument values:

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}(1)$$

The last possibility is not especially interesting for functions of one variable but we may be able to deduce strictness for functions of several variables by a combination of the two situations.

$$\mathbf{g}(\mathbf{x}, \mathbf{y}) = \textbf{if } \mathbf{y} = 0 \textbf{ then } \mathbf{x} \textbf{ else } \mathbf{g}(\mathbf{x} + 1, \mathbf{y} - 1)$$

This function is strict in both arguments. We can deduce strictness in the first argument without examining whether the function is total (and thus uses its argument). We know, however, that the function will either terminate and use its argument or fail to terminate. In both cases the function will be strict.

2.1 A lazy functional language

We will here use a functional language which is quite similar to the functional language presented in the domain theory notes. The only difference is that function calls will now be done by *call-by-need* or *call-by-name* rather than *call-by-value*. We cannot denotationally distinguish between *call-by-need* and *call-by-name* if programs have no side effects. The difference lies only in the number of times an expression is computed and not in possible values of programs or expression.

We will here describe the semantics for this small lazy functional language. This semantics will later be referred to as the *the standard semantics* or *the standard interpretation* as opposed to a rather special semantics we will see later. The word *standard* is a somewhat abused word but should here only be seen in contrast to the analysis we will construct later.

Semantics domains.

$$\begin{array}{lll} \mathbf{D} & = \mathbf{V}_\perp & \text{—values} \\ \Phi & = (\mathbf{D^k} \to \mathbf{D})^\mathbf{n} & \text{—function denotations} \end{array}$$

Semantics functions.

$$\begin{array}{ll} \mathsf{E}[\![\mathbf{exp}]\!] & : \Phi \to \mathbf{D^k} \to \mathbf{D} \\ \mathsf{P}[\![\mathbf{prog}]\!] & : \Phi \end{array}$$

Definition.

$$\begin{array}{ll} \mathsf{E}[\![\mathbf{c_i}]\!]\phi\nu & = \mathbf{const_i} \\ \mathsf{E}[\![\mathbf{x_i}]\!]\phi\nu & = \nu_\mathbf{i} \\ \mathsf{E}[\![\mathbf{a_i}(\mathbf{e}_1, \ldots, \mathbf{e_k})]\!]\phi\nu & = \mathsf{strict}\ \mathbf{basic_i}\langle \mathsf{E}[\![\mathbf{e}_1]\!]\phi\nu, \ldots, \mathsf{E}[\![\mathbf{e_k}]\!]\phi\nu\rangle \\ \mathsf{E}[\![\mathbf{if}\ \mathbf{e}_1\ \mathbf{then}\ \mathbf{e}_2\ \mathbf{else}\ \mathbf{e}_3]\!]\phi\nu & = \mathsf{cond}(\mathsf{E}[\![\mathbf{e}_1]\!]\phi\nu, \mathsf{E}[\![\mathbf{e}_2]\!]\phi\nu, \mathsf{E}[\![\mathbf{e}_3]\!]\phi\nu) \\ \mathsf{E}[\![\mathbf{f_i}(\mathbf{e}_1, \ldots, \mathbf{e_k})]\!]\phi\nu & = \phi_\mathbf{i}\langle \mathsf{E}[\![\mathbf{e}_1]\!]\phi\nu, \ldots, \mathsf{E}[\![\mathbf{e_k}]\!]\phi\nu\rangle \end{array}$$

$$\begin{array}{l} \mathsf{P}[\![\ \mathbf{f}_1(\mathbf{x}_1, \ldots, \mathbf{x_k}) = \mathbf{e}_1 \\ \qquad\qquad \vdots \\ \quad \mathbf{f_n}(\mathbf{x}_1, \ldots, \mathbf{x_k}) = \mathbf{e_n}]\!] = \mathsf{fix}\lambda\phi.\ \langle \mathsf{E}[\![\mathbf{e}_1]\!]\phi, \ldots, \mathsf{E}[\![\mathbf{e_n}]\!]\phi\rangle \end{array}$$

with

$$\mathsf{strict}\mathbf{f}\langle \mathbf{v}_1, \ldots, \mathbf{v_k}\rangle = \mathbf{if}\ \mathbf{v}_1 = \perp \vee \cdots \vee \mathbf{v_k} = \perp\ \mathbf{then}\ \perp\ \mathbf{else}\ \mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{v_k})$$

The semantics is very similar to the *call-by-value* semantics presented earlier. The only difference is that the function strict is no longer called at function calls.

2.2 Abstract domain

We will use a two-point domain to examine the strictness of functions. The domain is called $\mathbb{2}$ with the two elements 0 and 1 ordered by $0 \sqsubseteq 1$.

$$\mathbb{2} = \{0, 1\}, \qquad 0 \sqsubseteq 1$$

There are two interesting operations on the two-point domain: minimum and maximum. We will frequently use an infix notation for these operations:

$$\mathbf{d}_1 \wedge \mathbf{d}_2 = \mathbf{min}(\mathbf{d}_1, \mathbf{d}_2)$$
$$\mathbf{d}_2 \vee \mathbf{d}_2 = \mathbf{max}(\mathbf{d}_1, \mathbf{d}_2)$$

In domain theory we do not normally think of these operations as **min** and **max** but as respectively *greatest lower bound* (*infimum* or *glb*) and *least upper bound* (*supremum* or *lub*). For a domain as $\mathbb{2}$, however, this really is the same.

The domain $\mathbb{2}$ will be used to describe whether an element in $\mathbf{D}$ is *defined*, that is the bottom element $\perp$ or not. For this purpose we can define a so-called *abstraction function*

$$\alpha : \mathbf{D} \to \mathbb{2}$$
$$\alpha(\mathbf{d}) = \mathbf{if}\ \mathbf{d} = \perp\ \mathbf{then}\ 0\ \mathbf{else}\ 1$$

where it holds that $\forall \mathbf{d} \in \mathbf{D}.\ \mathbf{d} = \perp \Leftrightarrow \alpha(\mathbf{d}) = 0$. We will not define a *concretisation function* here. We do not need it and there is no obvious candidate which embeds the domain $\mathbb{2}$ in $\mathbf{D}$.

2.3 Strictness function

Now, let $\mathbf{f} : \mathbf{D^k} \to \mathbf{D}$ be a function whose strictness properties we would like to examine. Let us further assume that we have defined a function $\mathbf{f^\bullet} : \mathbb{2}^{\mathbf{k}} \to \mathbb{2}$ which satisfies

$$\forall \langle \mathbf{d}_1, \ldots, \mathbf{d_k} \rangle \in \mathbf{D^k}.\ \alpha(\mathbf{f}(\mathbf{d}_1, \ldots, \mathbf{d_k})) = \mathbf{f^\bullet}(\alpha(\mathbf{d}_1), \ldots, \alpha(\mathbf{d_k}))$$

If that is the case we then know that if

$$\mathbf{f^\bullet}(1, \ldots, 1, 0, 1, \ldots, 1) = 0$$

where all arguments are 1 except for the j'th, then $\mathbf{f}$ is strict in its j'th argument. This tells us that if all arguments, except the j'th, are well-defined, then the result will be undefined. The advantage with this description of strictness is that the function $\mathbf{f}^\bullet$ is finite. Each argument can only have two different values so we can tabulate the function by computing its value for all the possible arguments.

Undecidable. Unfortunately is it not always possible to construct a $\mathbf{f}^\bullet$ function. A function as

$$\mathbf{g}(\mathbf{x}, \mathbf{y}) = \mathbf{if}\ \mathbf{x} = 0\ \mathbf{then}\ \mathbf{y}\ \mathbf{else}\ 0$$

is strict in its first argument, but will only use its second argument if the first argument is 0. More generally the strictness property is *undecidable*, so even if a function always uses one of its arguments then we cannot expect a finite description (such as $\mathbf{f}^\bullet$) to show it.

Approximations. Since we cannot expect to construct a function such as $\mathbf{f}^\bullet$ which gives a precise description of the strictness of $\mathbf{f}$, then we can try and make two functions where one give too high values and the other too low values. This means that we would like to construct two functions which satisfy the following properties.

$$\forall \langle \mathbf{d}_1, \ \ldots, \mathbf{d_k} \rangle \in \mathbf{D^k}.\ \alpha(\mathbf{f}(\mathbf{d}_1, \ \ldots, \mathbf{d_k})) \sqsubseteq \mathbf{f}^\sharp(\alpha(\mathbf{d}_1), \ \ldots, \alpha(\mathbf{d_k}))$$

$$\forall \langle \mathbf{d}_1, \ \ldots, \mathbf{d_k} \rangle \in \mathbf{D^k}.\ \alpha(\mathbf{f}(\mathbf{d}_1, \ \ldots, \mathbf{d_k})) \sqsupseteq \mathbf{f}^\flat(\alpha(\mathbf{d}_1), \ \ldots, \alpha(\mathbf{d_k}))$$

The notation here is taken from the world of music where $\sharp$ (sharp) and $\flat$ (flat) respectively lift and lower a note by half a tone. We are always able to construct such functions since we may use the constant function 0 as a candidate for $\mathbf{f}^\flat$ and the constant function 1 as $\mathbf{f}^\sharp$. We are normally able to construct better approximations using a little bit of ingenuity.

We can also examine strictness of a function using these functions. We know that if

$$\mathbf{f}^\sharp(1, \ \ldots, 1, 0, 1, \ \ldots, 1) = 0$$

where all arguments are 1 except the j'th, then $\mathbf{f}$ is strict in its j'th argument. We can conclude this since $\mathbf{f}^\sharp$ gives an upper bound, and if the upper bound is the least element 0 then $\mathbf{f}$ must return the bottom element when its j'th argument is undefined. The function $\mathbf{f}^\sharp$ is clearly the most interesting of the two approximations when we want to analyse strictness. We will examine this function more closely in the rest of this section.

Examples. Let us now see how we can construct $\mathbf{f}^\sharp$ functions for some well-known functions.

Previously, we have used multiplication lifted to $\mathbb{N}_\perp$:

$$\mathsf{mul}_\perp : \mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$$
$$\mathsf{mul}_\perp(\mathbf{x}, \mathbf{y}) = \mathbf{if}\ \mathbf{x} = \perp \vee \mathbf{y} = \perp\ \mathbf{then}\ \perp\ \mathbf{else}\ \mathbf{x} * \mathbf{y}$$

The constant function 1 will, of course, be a safe candidate for $\mathsf{mul}^\sharp$, but we can do better since we can use

$$\mathsf{mul}^\sharp(\mathbf{x}, \mathbf{y}) = \mathbf{x} \wedge \mathbf{y} = \mathbf{min}(\mathbf{x}, \mathbf{y})$$

If one or both arguments of the multiplication is undefined (0) then we know that the result is undefined, otherwise the result is defined (1). This is precisely the minimum of the two values.

We have not discussed what happens in case of run-time errors such as overflow. A function like integer division can be undefined even if both arguments are defined. This does not give any problems since

$$\mathsf{div}^\sharp(\mathbf{x}, \mathbf{y}) = \mathbf{x} \wedge \mathbf{y}$$

is still a safe candidate as a $\sharp$-function, since even though $\mathsf{div}^\sharp(1, 1) = 1$ and $\mathsf{div}(7, 0) = \perp$ then it only tells us that the $\sharp$-function is an upper bound.

Another function we frequently use is cond. In this case the $\sharp$-function is a bit more complicated since we can use

$$\mathsf{cond}^\sharp(\mathbf{b}, \mathbf{x}, \mathbf{y}) = \mathbf{b} \wedge (\mathbf{x} \vee \mathbf{y})$$

To check this we only need to consider the various possible values of the arguments. We can also read the definition as follows: If the conditional expression should be defined then both the condition and at least one of the other arguments must be defined.

2.4 Strictness interpretation

We are now ready to present our first real abstract interpretation. The idea is that we will make a semantics for the language where the meanings have the same relation to the standard meaning as $\mathbf{f}^\sharp$ had to $\mathbf{f}$ above. We construct a semantics for the language, which computes in zeroes and ones. Really, this is quite straight forward. We just substitute all the $\mathbf{D}$'s in the semantics with the domain $\mathbb{2}$ and see what happens.

Semantic functions. The semantics uses two semantic functions, just as in the semantics that described the usual meaning. We will mark the semantic functions with a sharp ($\sharp$) so as to distinguish them from the standard semantics.

$$\begin{aligned}\mathbf{E}^\sharp[\![\mathbf{exp}]\!] &: (2^{\mathbf{k}} \to 2)^{\mathbf{n}} \to 2^{\mathbf{k}} \to 2\\ \mathbf{P}^\sharp[\![\mathbf{prog}]\!] &: (2^{\mathbf{k}} \to 2)^{\mathbf{n}}\end{aligned}$$

Definition.

$$\begin{aligned}&\mathbf{E}^\sharp[\![\mathbf{c_i}]\!]\phi\nu && = 1\\ &\mathbf{E}^\sharp[\![\mathbf{x_i}]\!]\phi\nu && = \nu_{\mathbf{i}}\\ &\mathbf{E}^\sharp[\![\mathbf{a_i}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\nu && = \mathbf{E}^\sharp[\![\mathbf{e}_1]\!]\phi\nu \wedge \cdots \wedge \mathbf{E}^\sharp[\![\mathbf{e_k}]\!]\phi\nu\\ &\mathbf{E}^\sharp[\![\mathbf{if\ e}_1\ \mathbf{then\ e}_2\ \mathbf{else\ e}_3]\!]\phi\nu && = \mathbf{E}^\sharp[\![\mathbf{e}_1]\!]\phi\nu \wedge (\mathbf{E}^\sharp[\![\mathbf{e}_2]\!]\phi\nu \vee \mathbf{E}^\sharp[\![\mathbf{e}_3]\!]\phi\nu)\\ &\mathbf{E}^\sharp[\![\mathbf{f_i}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\nu && = \phi_{\mathbf{i}}\langle \mathbf{E}^\sharp[\![\mathbf{e}_1]\!]\phi\nu,\ \ldots,\ \mathbf{E}^\sharp[\![\mathbf{e_k}]\!]\phi\nu\rangle\end{aligned}$$

$$\begin{aligned}\mathbf{P}^\sharp[\![\ &\mathbf{f}_1(\mathbf{x}_1,\ \ldots,\ \mathbf{x_k}) = \mathbf{e}_1\\ &\qquad\vdots\\ &\mathbf{f_n}(\mathbf{x}_1,\ \ldots,\ \mathbf{x_k}) = \mathbf{e_n}]\!] = \mathsf{fix}\lambda\phi.\ \langle \mathbf{E}^\sharp[\![\mathbf{e}_1]\!]\phi,\ \ldots,\ \mathbf{E}^\sharp[\![\mathbf{e_n}]\!]\phi\rangle\end{aligned}$$

So far so good. This, however, is only the first step since we must show that this semantics gives a safe description of the strictness of programs.

Correctness. We will show that the semantics function $\mathbf{P}^\sharp$ satisfies the following property.

$$\forall\rho, \mathbf{i} : \alpha((\mathbf{P}[\![\mathbf{p}]\!] \downarrow \mathbf{i})\rho) \sqsubseteq (\mathbf{P}^\sharp[\![\mathbf{p}]\!] \downarrow \mathbf{i})\langle\alpha(\rho_1),\ \ldots,\ \alpha(\rho_{\mathbf{k}})\rangle$$

The proof consists of two parts: a local part using structural induction and a global part using fixpoint induction. The proof is fairly easy but we will go through it in small steps since it is the first time we have a proof of this kind in the notes.

Proof. To a start we introduce a relation between real functions and strictness functions. Functions $\mathbf{f} : \mathbf{D^k} \to \mathbf{D}$ and $\mathbf{f^\sharp}\, 2^{\mathbf{k}} \to 2$ are related if they satisfy the following condition.

$$\mathbf{f}\ \mathbf{R}\ \mathbf{f^\sharp} \Leftrightarrow \forall \rho \in \mathbf{D^k}, \rho^\sharp \in 2^{\mathbf{k}}\,.\, \alpha(\rho_1) \sqsubseteq \rho_1^\sharp \wedge \cdots \wedge \alpha(\rho_{\mathbf{k}}) \sqsubseteq \rho_{\mathbf{k}}^\sharp \Rightarrow \alpha(\mathbf{f}(\rho)) \sqsubseteq \mathbf{f^\sharp}(\rho^\sharp)$$

The relation is inductive (limit preserving) since α is strict and that the relation here is the *logical* lifted relation to the function domain. We can, of course, also show this directly but there is no reason to do the work twice when we can get the result for free.

The condition we want to prove is that the two semantics for programs are related with respect to this relation.

$$\mathsf{P}[\![\mathbf{p}]\!] \downarrow \mathbf{i}\ \ \mathbf{R}\ \ \mathsf{P}^\sharp[\![\mathbf{p}]\!] \downarrow \mathbf{i}$$

for $\mathbf{i} = 1, \ldots, \mathbf{n}$ and all programs $\mathbf{p}$.

Local part. In the local part of the proof we will show the relation between the semantic functions E and $\mathsf{E}^\sharp$. We will now show that

$$\forall \phi \in (\mathbf{D^k} \to \mathbf{D})^{\mathbf{n}}, \phi^\sharp \in (2^{\mathbf{k}} \to 2)^{\mathbf{n}}.\ \phi_1\ \mathbf{R}\ \phi_1^\sharp \wedge \cdots \wedge \phi_{\mathbf{n}}\ \mathbf{R}\ \phi_{\mathbf{n}}^\sharp \Rightarrow \mathsf{E}[\![\mathbf{e}]\!]\phi\ \mathbf{R}\ \mathsf{E}^\sharp[\![\mathbf{e}]\!]\phi^\sharp$$

for all expression $\mathbf{e}$.

We will now prove the relation between $\mathsf{E}[\![\mathbf{e}]\!]$ and $\mathsf{E}^\sharp[\![\mathbf{e}]\!]$ by *structural induction* over possible expression. This means that we to a start will show that it holds for constants and parameters, and afterwards we consider composite expression under the assumption that it has been proved for the subexpressions. We have then argued that it holds for all expressions.
For constants $\mathbf{c_i}$ we should show that

$$\forall \rho \in \mathbf{D^k}, \rho^\sharp \in 2^{\mathbf{k}}\,.\, \alpha(\rho_1) \sqsubseteq \rho_1^\sharp \wedge \cdots \wedge \alpha(\rho_{\mathbf{k}}) \sqsubseteq \rho_{\mathbf{k}}^\sharp \Rightarrow \alpha(\mathbf{c_i}) \sqsubseteq 1$$

For parameters $\mathbf{x_i}$ we should show that

$$\forall \rho \in \mathbf{D^k}, \rho^\sharp \in 2^{\mathbf{k}}\,.\, \alpha(\rho_1) \sqsubseteq \rho_1^\sharp \wedge \cdots \wedge \alpha(\rho_{\mathbf{k}}) \sqsubseteq \rho_{\mathbf{k}}^\sharp \Rightarrow \alpha(\rho_{\mathbf{i}}) \sqsubseteq \rho_{\mathbf{i}}^\sharp$$

Both parts are trivially satisfied.
For standard operations $\mathbf{a_i}(\mathbf{e}_1, \ldots, \mathbf{e_k})$ we know that the relation holds for the subexpressions. Let ϕ, $\phi^\sharp$, ρ, $\rho^\sharp$ satisfy

$$\forall \mathbf{j}.\ \phi_{\mathbf{j}}\ \mathbf{R}\ \phi_{\mathbf{j}}^\sharp \qquad \text{and} \qquad \forall \mathbf{j}.\ \alpha(\rho_{\mathbf{j}}) \sqsubseteq \rho_{\mathbf{j}}^\sharp$$

We know that

$$\forall \mathbf{j}.\ \alpha(\mathsf{E}[\![\mathbf{e_j}]\!]\phi\rho) \sqsubseteq \mathsf{E}^\sharp[\![\mathbf{e_j}]\!]\phi^\sharp\rho^\sharp$$

and we should show that

$$\alpha(\mathsf{strict}\ \mathbf{basic_i}(\mathbf{v}_1,\ \ldots,\ \mathbf{v_k})) \sqsubseteq \mathbf{v}_1^\sharp \wedge \ldots \wedge_{\mathbf{k}}^\sharp$$

with

$$\mathbf{v_j} = \mathsf{E}[\![\mathbf{e_j}]\!]\phi\rho \qquad \text{and} \qquad \mathbf{v_j}^\sharp = \mathsf{E}^\sharp[\![\mathbf{e_j}]\!]\phi^\sharp\rho^\sharp$$

If all $\mathbf{v_j}^\sharp$ are 1 there is nothing to show. If, on the other hand, one of the values $\mathbf{v}_\ell^\sharp$ are equal to 0 we know that $\mathbf{v}_\ell = \bot$ and hence

$$\mathsf{strict}\ \mathbf{basic_i}(\mathbf{v}_1,\ \ldots,\ \mathbf{v_k}) = \bot$$

Consequently the relation is also satisfied for the standard operation. The proof for the conditional expression follows a similar pattern.
For function calls $\mathbf{f_i}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})$ we assume that the condition has been proved for subexpressions and thus that $\alpha(\mathbf{v_j}) \sqsubseteq \mathbf{v_j}^\sharp$ where $\mathbf{v_j} = \mathsf{E}[\![\mathbf{e_j}]\!]\phi\rho$ and $\mathbf{v_j}^\sharp = \mathsf{E}^\sharp[\![\mathbf{e_j}]\!]\phi^\sharp\rho^\sharp$. We should prove that

$$\alpha(\phi_{\mathbf{j}}(\mathbf{v}_1,\ \ldots,\ \mathbf{v_k}) \sqsubseteq \phi_{\mathbf{j}}^\sharp(\mathbf{v}_1^\sharp,\ \ldots,\ \mathbf{v_k}^\sharp)$$

but this follows from the assumption that ϕ and $\phi^\sharp$ are related.

Global part. The global part uses fixpoint induction to prove that the fixpoints

$$\mathsf{fix}\lambda\phi.\ \langle \mathsf{E}[\![\mathbf{e}_1]\!]\phi,\ \ldots,\ \mathsf{E}[\![\mathbf{e_n}]\!]\phi\rangle$$

and

$$\mathsf{fix}\lambda\phi^\sharp.\ \langle \mathsf{E}^\sharp[\![\mathbf{e}_1]\!]\phi^\sharp,\ \ldots,\ \mathsf{E}^\sharp[\![\mathbf{e_n}]\!]\phi^\sharp\rangle$$

are related. This follows directly from the inductiveness of the relation $\mathbf{R}$.

End of proof

Example. Let us now return to the example from the start of this section.

$$\mathbf{g}(\mathbf{x}, \mathbf{y}) = \textbf{if } \mathbf{y} = 0 \textbf{ then } \mathbf{x} \textbf{ else } \mathbf{g}(\mathbf{x} + 1, \mathbf{y} - 1)$$

The strictness interpretation gives us the following meaning

$$\mathbf{g}^\sharp(\mathbf{x}, \mathbf{y}) = \mathbf{y} \wedge (\mathbf{x} \vee \mathbf{g}^\sharp(\mathbf{x}, \mathbf{y}))$$

or more precisely

$$\mathbf{g}^\sharp(\mathbf{x}, \mathbf{y}) = \mathsf{fix}(\lambda\phi.\ \mathbf{y} \wedge (\mathbf{x} \vee \phi(\mathbf{x}, \mathbf{y})))$$

In the analysis of the strictness of the function we will compute $\mathbf{g}^\sharp(1, 0)$ and $\mathbf{g}^\sharp(0, 1)$. The first value is easy to find since

$$\mathbf{g}^\sharp(1, 0) = 0 \wedge (1 \vee \mathbf{g}^\sharp(1, 0)) = 0$$

since minimum of 0 and anything is 0. In it second case we have

$$\mathbf{g}^\sharp(0, 1) = 1 \wedge (0 \vee \mathbf{g}^\sharp(0, 1))$$

and seen as an equation it will both have $\mathbf{g}^\sharp(0, 1) = 0$ and $\mathbf{g}^\sharp(0, 1) = 1$ as solutions. We are interested in the least solution so we can conclude that $\mathbf{g}^\sharp(0, 1) = 0$ and thus that the function is strict in both arguments. Well, that was that.

2.5 Finding fixpoints

Strictness functions looks almost like our usual recursive functions but there is a difference. A function such as

$$\mathbf{h}(\mathbf{x}) = \mathbf{h}(\mathbf{x})$$

will not terminate for any arguments. The similar strictness function should certainly terminate and return the value 0 for all arguments.

In the example above we found the fixpoint of the equation defining $\mathbf{g}^\sharp$ using some reasoning about least elements etc. Although the discussion was quite straightforward it may sound difficult to automate and use in compilers.

We could, however, also have found the fixpoint as the limit of the ascending Kleene chain. We want to find a fixpoint in the domain $\mathbb{2}^2 \to \mathbb{2}$ and as usual we start with the least element and compute better approximations using the functional equation.

The least element in the domain $\mathbb{2}^2 \to \mathbb{2}$ is the constant function 0 since 0 is the least element in $\mathbb{2}$ The elements in the ascending Kleene chain are then

$$\mathbf{g}_0^\sharp = [(0,0) \mapsto 0, (1,0) \mapsto 0, (0,1) \mapsto 0, (1,1) \mapsto 0]$$
$$\mathbf{g}_1^\sharp = [(0,0) \mapsto 0, (1,0) \mapsto 0, (0,1) \mapsto 0, (1,1) \mapsto 1]$$
$$\mathbf{g}_2^\sharp = [(0,0) \mapsto 0, (1,0) \mapsto 0, (0,1) \mapsto 0, (1,1) \mapsto 1]$$

We reached the fixpoint quite fast although it did require some more computation compared to the more intuitive reasoning above. We do, however, know that we always will be able to compute these strictness description and that the computation will terminate. It is therefore safe to use such analyses in a compiler without the risk that the compiler will not stop. The fixpoint can be found in many ways but we know that it always can be done.

Complexity. We have argued that a strictness analysis will terminate. We know that the analysis is described as a fixpoint over a finite domain since there are $2^{\mathbf{k}}$ elements in the set $\mathbb{2}^{\mathbf{k}}$. In the set $\mathbb{2}^{\mathbf{k}} \to \mathbb{2}$ there are $2^{(2^{\mathbf{k}})}$ elements and of these not all are continuous. Each of these can be described by $2^{\mathbf{k}}$ different function values and a chain of such function can at most contain $2^{\mathbf{k}} + 1$ different elements. This means that an implementation will terminate and we may say that the analysis is *decidable*.

It is not really a requirement that the domain is finite. We only require that there are no chains with infinitely many different values in the domain. Domains with this property are often referred to satisfying *the ascending chain condition*.

These calculations of the number of iterations show a problem with strictness analysis. If there are many parameters to functions then the maximal number iterations and function results which should be saved and tested, will be rather large. We need to evaluate $2^{\mathbf{k}}$ values up to $2^{\mathbf{k}} + 1$ times where $\mathbf{k}$ is the number of arguments to the function. Now, we rarely write function with 50 or 100 arguments but it could happen and the the compiler may then need hundreds of years and lots of hard disks to complete the analysis. Programs generated by other programs frequently have a form that surprise compiler writers. The alternative is to find more clever ways to compute fixpoints. A simple iteration of values until convergence is the direct implementation, but depending of the situation one can use depth-first computations, tree structure representation of function values or something else. We will return to this in a later section.

3 Higher-order strictness analysis

Last section considered strictness analysis of a first-order functional language. In this section we will extend that method to higher-order typed languages.

3.1 Language

Consider a language of recursion equation system with higher order functions.

$$\begin{aligned} \mathbf{p} \quad &: \mathbf{f}_1 = \mathbf{e}_1 : \tau_1 \\ &\vdots \\ &\mathbf{f_n} = \mathbf{e_n} : \tau_\mathbf{n} \end{aligned}$$

$\mathbf{e}$	$: \mathbf{x_i}$	Parameters
	$\mid \mathbf{c_i}$	Constants
	$\mid \mathbf{f_i}$	Functions
	$\mid \lambda \mathbf{x_j} : \tau_\mathbf{j}.\ \mathbf{e}$	Abstraction
	$\mid \mathbf{e}_1(\mathbf{e}_2)$	Application

Expressions are assumed to be strongly (monomorphically) typed as functions or base values.

$$\begin{aligned} \tau \quad &: \tau_1 \rightarrow \tau_2 \\ &\mid \mathbf{Base} \end{aligned}$$

3.2 Standard interpretation

The semantics uses a recursively defined domain of values

$$\mathbf{D} \simeq \mathbf{Base}_\perp + \mathbf{D} \to \mathbf{D}$$

where **Base** is a set of base values. We also need the following semantic domains.

$\mathbf{Var} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots\}$	Parameters
$\rho \in \mathbf{Var} \to \mathbf{D}$	Parameter environment
$\phi \in \Phi = \mathbf{D^n}$	Function environment

Using these definitions the type of the semantic functions are as follows.

$$\mathsf{M}[\![\mathbf{p}]\!] : \mathbf{D^n}$$
$$\mathsf{E}[\![\mathbf{e}]\!] : \mathbf{D^n} \to (\mathbf{Var} \to \mathbf{D}) \to \mathbf{D}$$

$$\begin{aligned}
\mathsf{E}[\![\mathbf{x_i}]\!]\phi\rho &= \rho(\mathbf{x_i}) \\
\mathsf{E}[\![\mathbf{c_i}]\!]\phi\rho &= \mathbf{const_i} \\
\mathsf{E}[\![\mathbf{f_i}]\!]\phi\rho &= \phi_\mathbf{i} \\
\mathsf{E}[\![\lambda \mathbf{x_j} : \tau_\mathbf{j}.\ \mathbf{e}]\!]\phi\rho &= \lambda \mathbf{v}.\ \mathsf{E}[\![\mathbf{e}]\!]\phi\rho[\mathbf{x_j} \mapsto \mathbf{v}] \\
\mathsf{E}[\![\mathbf{e}_1(\mathbf{e}_2)]\!]\phi\rho &= (\mathsf{E}[\![\mathbf{e}_1]\!]\phi\rho)(\mathsf{E}[\![\mathbf{e}_2]\!]\phi\rho) \\
\mathsf{M}[\![\mathbf{f}_1 = \mathbf{e}_1 : \tau_1 \cdots \mathbf{f_n} = \mathbf{e_n} : \tau_\mathbf{n}]\!] &= \mathbf{fix}\lambda\phi.\ \langle \mathsf{E}[\![\mathbf{e}_1]\!]\phi\rho_0, \ \ldots, \mathsf{E}[\![\mathbf{e_n}]\!]\phi\rho_0 \rangle \\
&\quad \text{where } \rho_0 = \lambda \mathbf{x}.\ \perp_\mathbf{D}
\end{aligned}$$

with constants $\mathbf{const_i} \in \mathbf{D}$ for constants $\mathbf{c_i}$ in the program.

Notice that this semantics may also be used if the language is not strongly typed. The strictness analysis we will construct here will, however, only be finite if the language is typed.

3.3 Abstraction

The purpose of the strictness analysis is to give a safe description of the termination properties of the functions. The safety condition is quite simple in the first-order case where functions either terminate or fail to terminate. In the higher-order case, a function may return a function which is strict in some arguments but not in others.

Before we define the analysis we want to describe what the best possible approximation is. We may then prove that the strictness analysis is a safe approximation to this best approximation. The difference is that the strictness analysis is decidable (computable) whereas this best approximation is not.

The best approximation is constructed recursively using an abstraction function. If a value is of base type (**Base**) then the termination properties may be described using the abstraction function from first-order strictness analysis.

$$\begin{aligned}&\alpha_{\mathbf{Base}} : \mathbf{Base}_\perp \to 2\\ &\alpha_{\mathbf{Base}}(\mathbf{d}) = \mathbf{if}\ \mathbf{d} = \perp\ \mathbf{then}\ 0\ \mathbf{else}\ 1\end{aligned}$$

where

$$2 = \{0, 1\}, \qquad 0 \sqsubseteq 1$$

For a function $\mathbf{f} : \mathbf{Base}_\perp \to \mathbf{Base}_\perp$ on base values we may construct a best approximation as follows

$$\begin{aligned}&\alpha_{\mathbf{Base}\to\mathbf{Base}} : (\mathbf{Base}_\perp \to \mathbf{Base}_\perp) \to (2 \to 2)\\ &\alpha_{\mathbf{Base}\to\mathbf{Base}}(\mathbf{f}) = \lambda\mathbf{x}.\ \ \mathbf{if}\ \mathbf{x} = 0\ \mathbf{then}\ \alpha_{\mathbf{Base}}(\mathbf{f}(\perp))\\ &\qquad\qquad\qquad\qquad\ \mathbf{else}\ \mathbf{max}\{\alpha_{\mathbf{Base}}(\mathbf{f}(\mathbf{y})) \mid \mathbf{y} \in \mathbf{Base}\}\end{aligned}$$

Remember that in strictness analysis we want to detect guaranteed non-termination. Only if $\mathbf{f}$ fails to terminate for all defined arguments should $\alpha_{\mathbf{Base}\to\mathbf{Base}}(\mathbf{f})(1)$ return 0.

Now let $\mathbf{D_t}$ be the subset of $\mathbf{D}$ of objects with type $\mathbf{t}$ and let $\mathbf{B_t}$ be the domain of strictness functions for type $\mathbf{t}$. The generalisation to more complex types is

$$\begin{aligned}&\alpha_{\mathbf{t}_1\to\mathbf{t}_2} : \mathbf{D}_{\mathbf{t}_1\to\mathbf{t}_2} \to \mathbf{B}_{\mathbf{t}_1\to\mathbf{t}_2}\\ &\alpha_{\mathbf{t}_1\to\mathbf{t}_2}(\mathbf{f}) = \lambda\mathbf{x}.\ \bigsqcup\{\alpha_{\mathbf{t}_2}(\mathbf{f}(\mathbf{y})) \mid \alpha_{\mathbf{t}_1}(\mathbf{y}) \sqsubseteq \mathbf{x}\}\end{aligned}$$

where we have written $\bigsqcup$ instead of **max** and used $\sqsubseteq$ instead of equality. The latter should not change anything but it makes it easier to prove that the abstraction function is continuous.

Lemma. The central property of the abstraction function is

$$\forall \mathbf{x} : \mathbf{t}_1.\ \alpha_{\mathbf{t}_2}(\mathbf{f}(\mathbf{x})) \sqsubseteq \alpha_{\mathbf{t}_1 \to \mathbf{t}_2}(\mathbf{f})(\alpha_{\mathbf{t}_1}(\mathbf{x}))$$

Lemma. The abstraction function $\alpha_{\mathbf{t}}$ is continuous for all types $\mathbf{t}$.

Comments. As the abstraction function is defined recursively on the type structure, it requires the argument to be strongly typed. This is the reason why we require the language to be strongly typed.

We will omit the type index to the abstraction function when the type is clear from the context.

3.4 Strictness interpretation

The only difference between the strictness interpretation and the standard interpretation is for constants $\mathbf{c_i}$.

$$\begin{aligned}
\mathsf{E}^\sharp[\![\mathbf{x_i}]\!]\phi\rho &= \rho(\mathbf{x_i}) \\
\mathsf{E}^\sharp[\![\mathbf{c_i}]\!]\phi\rho &= \mathbf{const}_\mathbf{i}^\sharp \\
\mathsf{E}^\sharp[\![\mathbf{f_i}]\!]\phi\rho &= \phi_\mathbf{i} \\
\mathsf{E}^\sharp[\![\lambda \mathbf{x_j} : \tau_\mathbf{j}.\ \mathbf{e}]\!]\phi\rho &= \lambda \mathbf{v}.\ \mathsf{E}^\sharp[\![\mathbf{e}]\!]\phi\rho[\mathbf{x_j} \mapsto \mathbf{v}] \\
\mathsf{E}^\sharp[\![\mathbf{e}_1(\mathbf{e}_2)]\!]\phi\rho &= (\mathsf{E}^\sharp[\![\mathbf{e}_1]\!]\phi\rho)(\mathsf{E}^\sharp[\![\mathbf{e}_2]\!]\phi\rho)
\end{aligned}$$

where

$$\alpha(\mathbf{const_i}) \sqsubseteq \mathbf{const}_\mathbf{i}^\sharp$$

Theorem. Given ϕ, ϕ', ρ, ρ' such that

$$\alpha(\phi_{\mathbf{i}}) \sqsubseteq \phi'_{\mathbf{i}}, \quad \alpha(\rho(\mathbf{v})) \sqsubseteq \rho'(\mathbf{v}) \quad \forall \mathbf{i}, \mathbf{v}$$

we should prove that

$$\alpha(\mathsf{E}[\![\mathbf{e}]\!]\phi\rho) \sqsubseteq \mathsf{E}^\sharp[\![\mathbf{e}]\!]\phi'\rho'$$

The proof is by structural induction over expressions. Assume it has been proved for subexpressions, then prove:

$$\begin{aligned}
\alpha(\mathsf{E}[\![\lambda \mathbf{x_j} : \tau_\mathbf{j}.\ \mathbf{e}]\!]\phi\rho) &= \alpha(\lambda \mathbf{v}.\ \mathsf{E}[\![\mathbf{e}]\!]\phi\rho[\mathbf{x_j} \mapsto \mathbf{v}]) \\
&= \lambda \mathbf{x}.\ \bigsqcup\{\alpha(\mathsf{E}[\![\mathbf{e}]\!]\phi\rho[\mathbf{x_j} \mapsto \mathbf{y}]) \mid \alpha(\mathbf{y}) \sqsubseteq \mathbf{x}\} \\
&\sqsubseteq \lambda \mathbf{x}.\ \mathsf{E}^\sharp[\![\mathbf{e}]\!]\phi'\rho'[\mathbf{x_j} \mapsto \mathbf{x}] \\
&= \mathsf{E}^\sharp[\![\lambda \mathbf{x_j} : \tau_\mathbf{j}.\ \mathbf{e}]\!]\phi'\rho'
\end{aligned}$$

and

$$\begin{aligned}
\alpha(\mathsf{E}[\![\mathbf{e}_1(\mathbf{e}_2)]\!]\phi\rho) &= \alpha((\mathsf{E}[\![\mathbf{e}_1]\!]\phi\rho)(\mathsf{E}[\![\mathbf{e}_2]\!]\phi\rho)) \\
&\sqsubseteq \alpha(\mathsf{E}[\![\mathbf{e}_1]\!]\phi\rho)(\alpha(\mathsf{E}[\![\mathbf{e}_2]\!]\phi\rho)) \\
&\sqsubseteq (\mathsf{E}^\sharp[\![\mathbf{e}_1]\!]\phi'\rho')(\mathsf{E}^\sharp[\![\mathbf{e}_2]\!]\phi'\rho') \\
&= \mathsf{E}^\sharp[\![\mathbf{e}_1(\mathbf{e}_2)]\!]\phi'\rho'
\end{aligned}$$

Theorem. Define

$$\begin{aligned}
&\psi = \mathsf{M}[\![\mathbf{f}_1 = \mathbf{e}_1 : \tau_1 \cdots \mathbf{f_n} = \mathbf{e_n} : \tau_\mathbf{n}]\!] = \mathbf{fix}\lambda\phi.\ \langle \mathsf{E}[\![\mathbf{e}_1]\!]\phi\rho_0,\ \ldots,\ \mathsf{E}[\![\mathbf{e_n}]\!]\phi\rho_0\rangle \\
&\qquad \text{where } \rho_0 = \lambda \mathbf{x}.\ \bot_\mathbf{D}
\end{aligned}$$

$$\begin{aligned}
&\psi' = \mathsf{M}^\sharp[\![\mathbf{f}_1 = \mathbf{e}_1 : \tau_1 \cdots \mathbf{f_n} = \mathbf{e_n} : \tau_\mathbf{n}]\!] = \mathbf{fix}\lambda\phi'.\ \langle \mathsf{E}^\sharp[\![\mathbf{e}_1]\!]\phi'\rho'_0,\ \ldots,\ \mathsf{E}^\sharp[\![\mathbf{e_n}]\!]\phi'\rho'_0\rangle \\
&\qquad \text{where } \rho'_0 = \lambda \mathbf{x}.\ 0
\end{aligned}$$

then

$$\forall \mathbf{i}.\ \alpha(\psi_\mathbf{i}) \sqsubseteq \psi'_\mathbf{i}$$

The proof is by fixpoint induction using that the relation $\alpha(\phi) \sqsubseteq \phi'$ is inductive (preserve limits).

3.5 Power domains

The safety of the strictness analysis is here described using only an abstraction function. We have established a relationship of the form $\alpha(\psi_{\mathbf{i}}) \sqsubseteq \psi'_{\mathbf{i}}$. We can prove that this relationship is correct but we have not proved that α or $\mathsf{E}[\![\mathbf{e}]\!]$ are correct. Somehow α is our definition of what we mean by non-termination and $\mathsf{E}[\![\mathbf{e}]\!]$ is the meaning of the programs and as such, they are the foundation or axioms of the theory. The use of the abstraction function to explain what we mean by non-termination may feel a bit unnatural as it maps values into the abstract meanings. Another, maybe more natural, approach could be to define which real program behaviours are described by a strictness function. We can then argue that for all these programs that certain transformations are allowed. This would require the definition of a so-called concretisation function:

$$\gamma_{\mathbf{t}} : \mathbf{B_t} \to \mathcal{P}(\mathbf{D_t})$$
$$\gamma_{\mathbf{t}}(\mathbf{b}) = \{\mathbf{f} \mid \alpha(\mathbf{f}) = \mathbf{b}\}$$

This function, however, is not continuous so we can not prove a relationship of the form

$$\psi_{\mathbf{i}} \in \gamma(\psi'_{\mathbf{i}})$$

using fixpoint induction. Instead we may achieve continuity with a slight change in the definition:

$$\gamma_{\mathbf{t}}(\mathbf{b}) = \{\mathbf{f} \mid \alpha(\mathbf{f}) \sqsubseteq \mathbf{b}\}$$

This concretisation function will map strictness functions to downwards closed sets of real function. The set of downwards closed subsets of a domain is normally called the Hoare power domain.

3.6 Examples

Top and bottom. In the higher-order case the strictness analysis may need functional arguments and produce functional results. How do we then check whether a function is top element or bottom in the domain? Well, there is a simple method to do that since strictness functions are monotonic:

Define **top** and **bot** recursively as

$$\begin{aligned}
\mathbf{top}_2 &= 1 \\
\mathbf{top}_{\mathbf{u}\to\mathbf{t}} &= \lambda\mathbf{x}.\ \mathbf{top}_{\mathbf{t}} \\
\mathbf{bot}_2 &= 0 \\
\mathbf{bot}_{\mathbf{u}\to\mathbf{t}} &= \lambda\mathbf{x}.\ \mathbf{bot}_{\mathbf{t}}
\end{aligned}$$

Check whether a value is bottom or top:

$$\begin{array}{llll} \mathbf{x}_2 & = \mathbf{top}_2 & \Leftrightarrow \mathbf{x}_2 & = 1 \\ \mathbf{x}_{\mathbf{u}\to\mathbf{t}} & = \mathbf{top}_{\mathbf{u}\to\mathbf{t}} & \Leftrightarrow \mathbf{x}(\mathbf{bot}_\mathbf{u}) & = \mathbf{top}_\mathbf{t} \\ \mathbf{x}_2 & = \mathbf{bot}_2 & \Leftrightarrow \mathbf{x}_2 & = 0 \\ \mathbf{x}_{\mathbf{u}\to\mathbf{t}} & = \mathbf{bot}_{\mathbf{u}\to\mathbf{t}} & \Leftrightarrow \mathbf{x}(\mathbf{top}_\mathbf{u}) & = \mathbf{bot}_\mathbf{t} \end{array}$$

Example. Now consider this small program:

$$\mathbf{g\ f\ x\ y} = \mathbf{x} + \mathbf{f}(\mathbf{y})$$
$$\mathbf{h\ u} = \mathbf{g}(\lambda\mathbf{u}.\ \mathbf{v} * \mathbf{u})\ 5\ 4$$

The strictness versions are:

$$\mathbf{g}^\sharp\ \mathbf{f\ x\ y} = \mathbf{x} \wedge \mathbf{f}(\mathbf{y})$$
$$\mathbf{h}^\sharp\ \mathbf{u} = \mathbf{g}^\sharp(\lambda\mathbf{v}.\ \mathbf{v} \wedge \mathbf{u})\ 1\ 1$$

The strictness analysis will then consider the following calls of the strictness functions.

$$\mathbf{g}^\sharp(\lambda\mathbf{z}.\ 0)\ 1\ 1 = 0$$
$$\mathbf{g}^\sharp(\lambda\mathbf{z}.\ 1)\ 0\ 1 = 0$$
$$\mathbf{g}^\sharp(\lambda\mathbf{z}.\ 1)\ 1\ 0 = 1$$
$$\mathbf{h}\ 0 = \mathbf{g}^\sharp(\lambda\mathbf{v}.\ 0)\ 1\ 1 = 0$$

Example. Original program

$$\begin{array}{lcl} \mathbf{hof\ g\ x\ y} & = & \mathbf{g}(\mathbf{hof}(\mathbf{k}(0), \mathbf{x}, \mathbf{y} - 1)) + \\ & & \mathbf{if\ y} = 0\ \mathbf{then\ x\ else\ hof}(\mathbf{i}, 3, \mathbf{y} - 1) \\ \mathbf{k\ x\ y} & = & \mathbf{x}; \\ \mathbf{i\ x} & = & \mathbf{x}; \end{array}$$

The strictness versions are

$$\begin{array}{l} \mathbf{hof\ g\ x\ y} = \\ \quad \mathbf{g}(\mathbf{hof}(\mathbf{k}(1), \mathbf{x}, \mathbf{y})) \wedge \mathbf{y} \wedge (\mathbf{x} \vee \mathbf{hof}(\mathbf{i}, 1, \mathbf{y})); \\ \mathbf{k\ x\ y} = \mathbf{x}; \\ \mathbf{i\ x} = \mathbf{x}; \\ \mathbf{top\ x} = 1; \\ \mathbf{bot\ x} = 0; \end{array}$$

The strictness analysis will then consider the following calls of the strictness functions.

$$\begin{aligned}
&\mathbf{hof}(\mathbf{bot}(), 1, 1) &&= 0\\
&\mathbf{hof}(\mathbf{top}(), 0, 1) &&= 1\\
&\mathbf{hof}(\mathbf{top}(), 1, 0) &&= 0\\
&\mathbf{k}(0, 1) &&= 0\\
&\mathbf{k}(1, 0) &&= 1\\
&\mathbf{i}(0) &&= 0
\end{aligned}$$

4 Live variable analysis

Live variable analysis is one of the traditional data-flow analysis problem. It is normally stated for imperative programs, often in a flow-chart form. Abstract Interpretation was introduced to show that such analyses could be given a semantic basis. We will here present the live variable analysis as an abstract interpretation of an imperative language.

4.1 A small language

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

The language will be introduced with a context-free grammar and an informal description of its semantics. A formal semantics for the language will be given later.

$$
\begin{array}{lll}
\textbf{program} & ::= & \textbf{stmt} \\
\textbf{stmt} & ::= & \ell_{\mathbf{i}} \textbf{ : stmt} \\
 & | & \textbf{stmt ; stmt} \\
 & | & \textbf{goto } \ell_{\mathbf{i}} \\
 & | & \mathbf{x_i} \textbf{ := exp} \\
 & | & \textbf{if exp then stmt else stmt} \\
\textbf{exp} & ::= & \textbf{exp} + \textbf{exp} \\
 & | & \textbf{exp} - \textbf{exp} \\
 & | & \mathbf{c_i} \\
 & | & \mathbf{x_i}
\end{array}
$$

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

The output from the program is the contents of a given variable (say "$\mathbf{x}_0$") at the end of the execution. This means that a program will end with an implicit **print**($\mathbf{x}_0$) command. Another possibility would have been to let the values of all variables be available at the end of the program but this would make a live-variable analysis less interesting.

4.2 Standard interpretation

This section introduces the standard interpretation for the language. In practice the design of the standard interpretation will often go hand in hand with the abstract interpretation and the soundness proof. The general idea in the interpretations is to use a backward or continuation style semantics such that the meaning of a statement describes "the rest of the computation". In the case of the standard interpretation it is a mapping of the environment at the start of the statement to the final answer. For the abstract interpretation it is the set of variables which may be used in this or later statements along possible execution paths. A variable is said to be *live* at a given program point if it may be used in any execution path before it is assigned a new value. The relationship between these two interpretations can informally be stated as: if a variable is not live (*ie.* dead) then the standard interpretation will not change if that variable is made undefined. We will formalise this in section 4.4.

4.2.1 Semantic framework

We could now just have specified the standard semantics of the language as a denotational semantics and afterwards constructed the abstract interpretation. We will instead separate each interpretation into two parts where the first part will be common to the two interpretation. This first part is a skeleton semantics where certain symbols are left uninterpreted and an interpretation should then assign a meaning to these symbols. A skeleton semantics of this form is often referred to as a *semantic framework*.

The semantic framework will show the flow of information in the program but will leave the actual interpretation unspecified. The framework defines a continuation style (or backward) semantics. Meanings of statements are continuations: mappings of the state before the statement to a final result. The framework will use a *label environment* to keep the meanings of labels. The label environment is a mapping of label names to the continuation where the label appear in the program. The semantic function $\mathbf{S}$ takes two arguments $\mathbf{c}$ and ξ, where $\mathbf{c}$ is the continuation after the statement and ξ is the label environment. The result will be a tuple with the continuation from the state before the statement as first component and a label environment of labels that appear in the statement as second component.

$$
\begin{aligned}
&\mathbf{P}[\![\mathbf{program}]\!] && = \mathsf{letrec}\langle \mathbf{r}, \xi\rangle = \mathbf{S}[\![\mathbf{program}]\!]\mathbf{Final}\ \xi\ \mathsf{in}\ \xi \\
&\mathbf{S}[\![\ell_{\mathbf{i}} : \mathbf{stmt}]\!]\mathbf{c}\ \xi && = \mathsf{let}\langle \mathbf{r}, \rho\rangle = \mathbf{S}[\![\mathbf{stmt}]\!]\mathbf{c}\ \xi\ \mathsf{in}\ \langle \mathbf{r}, \rho[\ell_{\mathbf{i}} \mapsto \mathbf{r}]\rangle \\
&\mathbf{S}[\![\mathbf{stmt}_1\ ;\ \mathbf{stmt}_2]\!]\mathbf{c}\ \xi && = \mathsf{let}\langle \mathbf{r}_2, \rho_2\rangle = \mathbf{S}[\![\mathbf{stmt}_2]\!]\mathbf{c}\ \xi\ \mathsf{and}\ \langle \mathbf{r}_1, \rho_1\rangle = \mathbf{S}[\![\mathbf{stmt}_1]\!]\mathbf{r}_2\ \xi
\end{aligned}
$$

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad\quad \mathsf{in}\langle \mathbf{r}_1, \mathbf{Join}(\rho_1, \rho_2)\rangle \\
\mathbf{S}[\![\mathbf{goto}\ \ell_\mathbf{i}]\!]\mathbf{c}\ \xi &= \langle \mathbf{Goto}(\xi, \ell_\mathbf{i}), \mathbf{Nil}\rangle \\
\mathbf{S}[\![\mathbf{x_i} := \mathbf{exp}]\!]\mathbf{c}\ \xi &= \langle \mathbf{Update}(\mathbf{c}, \mathbf{x_i}, \mathbf{E}[\![\mathbf{exp}]\!]), \mathbf{Nil}\rangle \\
\mathbf{S}[\![\mathbf{if\ exp\ then\ stmt}\ \ \mathbf{else\ stmt}]\!]\mathbf{c}\ \xi &= \mathsf{let}\langle \mathbf{r}_1, \rho_1\rangle = \mathbf{S}[\![\mathbf{stmt}_1]\!]\mathbf{c}\ \xi\ \mathsf{and}\ \langle \mathbf{r}_2, \rho_2\rangle = \mathbf{S}[\![\mathbf{stmt}_2]\!]\mathbf{c}\ \xi \\
&\qquad\qquad\qquad\qquad\quad \mathsf{in}\langle \mathbf{If}(\mathbf{E}[\![\mathbf{exp}]\!], \mathbf{r}_1, \mathbf{r}_2), \mathbf{Join}(\rho_1, \rho_2)\rangle \\
\mathbf{E}[\![\mathbf{exp}_1 + \mathbf{exp}_2]\!] &= \mathbf{Add}(\mathbf{E}[\![\mathbf{exp}_1]\!], \mathbf{E}[\![\mathbf{exp}_2]\!]) \\
\mathbf{E}[\![\mathbf{exp} - \mathbf{exp}]\!] &= \mathbf{Sub}(\mathbf{E}[\![\mathbf{exp}_1]\!], \mathbf{E}[\![\mathbf{exp}_2]\!]) \\
\mathbf{E}[\![\mathbf{c_i}]\!] &= \mathbf{Const}(\mathbf{c_i}) \\
\mathbf{E}[\![\mathbf{x_i}]\!] &= \mathbf{Var}(\mathbf{x_i})
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathbf{Goto}(\xi, \mathbf{l}) = \mathbf{if}\ \xi(\mathbf{l}) = \mathbf{Undef}\ \mathbf{then\ Final\ else}\ \xi(\mathbf{l}) \\
&\mathbf{Nil} = \lambda \mathbf{l}.\ \mathbf{Undef} \\
&\mathbf{Join}(\xi_1, \xi_2) = \lambda \mathbf{l}.\ \ \mathbf{if}\ \xi_1(\mathbf{l}) = \mathbf{Undef}\ \mathbf{then}\ \xi_2(\mathbf{l})\ \mathbf{else}\ \xi_1(\mathbf{l})
\end{aligned}
$$

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

The label environment is defined as a fixpoint. We have used a letrec expression but we could also have written the fixpoint directly:

$$\mathbf{P}[\![\mathbf{program}]\!] = \mathsf{snd}(\mathsf{fix}(\lambda\langle \mathbf{r}, \xi\rangle.\ \mathbf{S}[\![\mathbf{program}]\!]\mathbf{Final}\ \xi))$$

In the standard interpretation the fixpoint may be implemented using a letrec expression in a functional language but the abstract interpretation will require fixpoint iteration where the bottom element is not represented as undefined.

4.2.2 Types

The semantic framework can be seen as a polymorphic function definition. It is possible to type-check the scheme and give types to the operator symbols. Let **M**, **D**, **U**, **A**, **N** be type variables. The types of the semantic functions are then

$$\begin{aligned}
&\mathsf{P}[\![\mathbf{program}]\!] : \mathbf{M} \\
&\mathsf{S}[\![\mathbf{stmt}]\!] : (\mathbf{M} \times (\mathbf{A} \to (\mathbf{M} + \mathbf{U}))) \to (\mathbf{M} \times (\mathbf{A} \to (\mathbf{M} + \mathbf{U}))) \\
&\mathsf{E}[\![\mathbf{exp}]\!] : \mathbf{D}
\end{aligned}$$

The types of the defined symbols are

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

and the types for the operator symbols are

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

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

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

4.2.3 Standard interpretation

In the standard interpretation the meaning (of type **M**) of a statement will be a mapping of the environment before the statement to the final result. The final result is the value of the variable "$\mathbf{x}_0$" at the end of the program.

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

The standard interpretation uses the following interpretation of type names:

$$\begin{aligned}&\mathbf{M} : \mathbf{env} \to \mathbb{Z}\\ &\mathbf{D} : \mathbf{env} \to \mathbb{Z}\\ &\mathbf{env} : \mathbb{A} \to \mathbb{Z}\end{aligned}$$

and the interpretation of operator symbols are

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

4.3 Live-variable analysis

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

The interpretations of the type names are

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

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

$$
\begin{aligned}
\mathbf{Update}(\mathbf{Cn}, \mathbf{n}, \mathbf{e}) &= \mathbf{e} \cup (\mathbf{Cn} \backslash \{\mathbf{n}\}) \\
\mathbf{If}(\mathbf{e}, \mathbf{Ct}, \mathbf{Cf}) &= \mathbf{e} \cup \mathbf{Ct} \cup \mathbf{Cf} \\
\mathbf{Final} &= \{\text{“}\mathbf{x}_0\text{”}\} \\
\mathbf{Add}(\mathbf{e}1, \mathbf{e}2) &= \mathbf{e}1 \cup \mathbf{e}2 \\
\mathbf{Sub}(\mathbf{e}1, \mathbf{e}2) &= \mathbf{e}1 \cup \mathbf{e}2 \\
\mathbf{Const}(\mathbf{n}) &= \varnothing \\
\mathbf{Var}(\mathbf{v}) &= \{\mathbf{v}\}
\end{aligned}
$$

All these functions can easily be seen to be continuous.

4.4 Correctness proof of the abstract interpretation

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

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

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

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

4.4.1 Relation

The relation for $\mathbf{M}$ and $\mathbf{D}$ is the same and is defined as

$$\mathbf{c_s}\ \mathbb{D}_{\mathbf{M}}\ \mathbf{c_a} \Leftrightarrow \forall \mathbf{x}.\ \mathbf{x} \notin \mathbf{c_a} \Rightarrow \forall \mathbf{e}.\ \mathbf{c_s}(\mathbf{e}) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \bot])$$

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

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

4.4.2 Local correctness

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

We will prove that

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

or equivalently that

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

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

$$\mathbf{c_s}\ \mathbb{D}_{\mathbf{M}}\ \mathbf{c_a}, \mathbf{v_s} = \mathbf{v_a}, \mathbf{e_s}\ \mathbb{D}\ \mathbf{e_a}$$

We need to prove that

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

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

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

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

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

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

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

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

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

This is true by the assumption ($\mathbf{c_s}\, \mathbb{D}_{\mathbf{M}}\, \mathbf{c_a} \wedge \mathbf{x} \notin \mathbf{e_a}$).

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

4.4.3 Fixpoint induction

The final part of the proof is to show that the relation $\mathbb{D}_{\mathbf{M}}$ is inductive. That is to prove that for any chain $\mathbf{X} \subseteq \mathbb{D}_{\mathbf{M}}$ that $\bigsqcup \mathbf{X} \subseteq \mathbb{D}_{\mathbf{M}}$. This follows from the fact that $\mathbf{M_a}$ is finite (there is only a finite number of variables in a program) and that $\forall \mathbf{e}.\ \mathbf{c_s}(\mathbf{e}) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \bot])$ is an inductive predicate on $\mathbf{M_s}$. This follows again from the continuity of $\mathbf{c_s}$.

4.5 Example

Consider the program

$$\begin{array}{ll}
\ell_1 : & \mathbf{x}_1 := \mathbf{x}_2 - 1 \\
& \mathbf{x}_0 := \mathbf{x}_1 + 1 \\
& \textbf{if } \mathbf{x}_1 = 0 \textbf{ then goto } \ell_3 \textbf{ else goto } \ell_2 \\
\ell_2 & \mathbf{x}_2 := \mathbf{x}_1 - 1 \\
& \textbf{goto } \ell_1 \\
\ell_3 & \mathbf{x}_0 := \mathbf{x}_0 + 1
\end{array}$$

The live variable analysis will produce the following label environment of liveness information at each label:

$$\begin{array}{l}
\ell_1 : \{\mathbf{x}_2\} \\
\ell_2 : \{\mathbf{x}_1\} \\
\ell_3 : \{\mathbf{x}_0\}
\end{array}$$

The result is obtained by propagating liveness information backwards through the flow-chart.

4.6 Denotational abstract interpretation

The idea of using a semantic framework can be taken a step further. When we write a denotational semantics we define the meaning of expressions using a domain-theoretic meta-language. This meta-language consists of λ-abstraction, function application and a number of continuous functions such as $\mathsf{mul}_\perp$, cond, fix, fst, snd, isl, inl, outl, etc. If we gave these meta-language constructs an abstract meaning we would also have given any language with a denotational semantics an abstract meaning. Thus, if we are able to give the metalanguage an abstract interpretation we can then use this as an analysis of any language. The seems to be two main problems with this approach. The domain-theoretic metalanguage is very powerful and it is difficult to construct good analyses which may work for any language. The second problem is that it is not that obvious how to relate information about meta-language constructs to the original programs.

5 Constant propagation

The strictness analysis for first-order functions extracted properties of the functions in a finite form. The analysis was *compostional* or *modular* in the sense that a function could be analysed independently of the context in which it appeared. The strictness of a function is a denotational property since it can be expressed in terms of the denotation of functions. In this section we will consider an example of an analysis which cannot directly be expressed in this way. The example will be a *constant propagation* analysis which gives a safe (not too small) description of the possible arguments to functions in internal calls of a recursion equation system.

5.1 Language

We will here consider a language of recursion equations. A program consists of $\mathbf{n}$ functions each of $\mathbf{k}$ arguments.

$$
\begin{array}{lcll}
\mathbf{e} & ::= & \mathbf{c_i} & \text{constants} \\
 & | & \mathbf{x_i} & \text{parameters } 1 \leq \mathbf{i} \leq \mathbf{k} \\
 & | & \mathbf{a_i}(\mathbf{e}_1, \ldots, \mathbf{e_k}) & \text{basic operations} \\
 & | & \mathbf{f_j}(\mathbf{e}_1, \ldots, \mathbf{e_k}) & \text{function calls } 1 \leq \mathbf{j} \leq \mathbf{n}
\end{array}
$$

and programs **prog** have the form:

$$
\begin{array}{lcll}
\mathbf{prog} & ::= & \mathbf{f}_1(\mathbf{x}_1, \ldots, \mathbf{x_k}) & = & \mathbf{e}_1 \\
 & & & \vdots & \\
 & & \mathbf{f_n}(\mathbf{x}_1, \ldots, \mathbf{x_k}) & = & \mathbf{e_n}
\end{array}
$$

All functions are first order over some domain (chain-complete partial order) $\mathbf{D}$ with a normal order evaluation strategy for arguments to function calls. The domain $\mathbf{D}$ may be a flat cpo or it may be a reflexive domain describing lazy data structures. Expressions are constructed from a set of constants $\mathbf{c_i}$ and basic operations $\mathbf{a_i}$ using parameters and function calls. For each constant $\mathbf{c_i}$ there must be a corresponding element $\mathbf{const_i} \in \mathbf{D}$ and for each basic operation $\mathbf{a_i}$ there must be a continuous function $\mathbf{basic_i} : \mathbf{D^k} \rightarrow_\mathbf{c} \mathbf{D}$. We may include a conditional function as a basic operation; in this case we may expect the domain $\mathbf{D}$ to contain the boolean values true and false.

Standard semantics. The usual fixpoint semantics for this type of language may be defined as follows. The semantics will use the following domains

$$\begin{array}{lll} \mathbf{d} \in & \mathbf{D} & \text{values} \\ \nu \in & \mathbf{D^k} & \text{parameter environment} \\ \phi \in & \Phi \quad = (\mathbf{D^k} \to_\mathbf{c} \mathbf{D})^\mathbf{n} & \text{function environment} \end{array}$$

There are two semantic functions:

$$\begin{array}{ll} \mathsf{E}[\![\mathbf{e}]\!] & : \Phi \to_\mathbf{c} \mathbf{D^k} \to_\mathbf{c} \mathbf{D} \\ \mathsf{U}[\![\mathbf{prog}]\!] & : \Phi \end{array}$$

with the definitions

$$\begin{array}{lcl} \mathsf{E}[\![\mathbf{c_i}]\!]\phi\nu & = & \mathbf{const_i} \\ \mathsf{E}[\![\mathbf{x_i}]\!]\phi\nu & = & \nu_\mathbf{i} \\ \mathsf{E}[\![\mathbf{a_j}(\mathbf{e}_1, \ldots, \mathbf{e_k})]\!]\phi\nu & = & \mathbf{basic_j}(\mathsf{E}[\![\mathbf{e}_1]\!]\phi\nu, \ldots, \mathsf{E}[\![\mathbf{e_k}]\!]\phi\nu) \\ \mathsf{E}[\![\mathbf{f_j}(\mathbf{e}_1, \ldots, \mathbf{e_k})]\!]\phi\nu & = & \phi_\mathbf{j}(\mathsf{E}[\![\mathbf{e}_1]\!]\phi\nu, \ldots, \mathsf{E}[\![\mathbf{e_k}]\!]\phi\nu) \end{array}$$

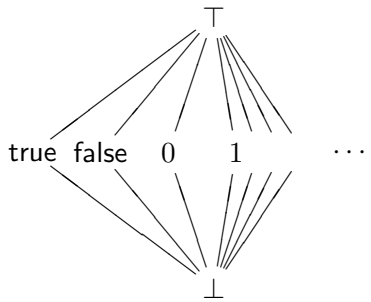
and

$$\mathsf{U}[\![\mathbf{f}_1(\mathbf{x}_1, \ldots, \mathbf{x_k}) = \mathbf{e}_1, \ldots, \mathbf{f_n}(\mathbf{x}_1, \ldots, \mathbf{x_k}) = \mathbf{e_n}]\!] = \mathbf{fix}\lambda\phi.\ \langle \mathsf{E}[\![\mathbf{e}_1]\!]\phi, \ldots, \mathsf{E}[\![\mathbf{e_n}]\!]\phi \rangle$$

The well-definedness of the fixpoint is guaranteed by the continuity of the semantic function E.

5.2 Constant propagation

The constant propagation should give a safe description of the possible arguments to the functions in a recursion equation system when only one of the functions can be called externally. In a constant propagation analysis we describe sets of possible arguments using the domain $\mathbf{D}^\top$ where elements in $\mathbf{D}$ describe singleton sets of values from $\mathbf{D}$ and the extra element $\top$ describe that an argument may have several different values. If an argument is described by a value in $\mathbf{D}$ then that argument in all internal calls must be a constant and the program can then be optimised accordingly.



The domain $\mathbf{D}^\top$

We can construct concretisation and abstraction functions between $\mathbf{D}^\top$

$$\begin{array}{lll} \alpha : \mathcal{P}(\mathbf{D}) \to \mathbf{D}^\top & & \gamma : \mathbf{D}^\top \to \mathcal{P}(\mathbf{D}) \\ \alpha(\varnothing) & = \bot_{\mathbf{D}} & \gamma(\top) = \mathbf{D} \\ \alpha(\{\mathbf{x}\}) & = \mathbf{x} & \gamma(\mathbf{x}) = \{\mathbf{x}\} \\ \alpha(\{\mathbf{x}_1,\ \ldots\}) & = \top & \end{array}$$

and $\mathcal{P}(\mathbf{D})$ as follows:
Basic operations on $\mathbf{D}$ can easily be extended to operations on this extended domain $\mathbf{D}^\top$.

$$\mathbf{basic_j^\top}(\mathbf{v}_1,\ \ldots,\ \mathbf{v_k}) = \mathbf{if\ some\ v_i} = \top\ \mathbf{then}\ \top\ \mathbf{else\ basic_j}(\mathbf{v}_1,\ \ldots,\ \mathbf{v_k})$$

Using this extension we may extend the semantics of programs to the domain $\mathbf{D}^\top$.

$$\begin{array}{lcl} \mathsf{E}^\top[\![\mathbf{c_i}]\!]\phi\nu & = & \mathbf{const_i} \\ \mathsf{E}^\top[\![\mathbf{x_i}]\!]\phi\nu & = & \nu_\mathbf{i} \\ \mathsf{E}^\top[\![\mathbf{a_j}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\nu & = & \mathbf{basic_j^\top}(\mathsf{E}^\top[\![\mathbf{e}_1]\!]\phi\nu,\ \ldots,\ \mathsf{E}^\top[\![\mathbf{e_k}]\!]\phi\nu) \\ \mathsf{E}^\top[\![\mathbf{f_j}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\nu & = & \phi_\mathbf{j}(\mathsf{E}^\top[\![\mathbf{e}_1]\!]\phi\nu,\ \ldots,\ \mathsf{E}^\top[\![\mathbf{e_k}]\!]\phi\nu) \end{array}$$

and

$$\mathsf{U}^\top[\![\mathbf{f}_1(\mathbf{x}_1,\ \ldots,\ \mathbf{x_k}) = \mathbf{e}_1,\ \ldots,\ \mathbf{f_n}(\mathbf{x}_1,\ \ldots,\ \mathbf{x_k}) = \mathbf{e_n}]\!] = \mathbf{fix}\lambda\phi.\ \langle \mathsf{E}^\top[\![\mathbf{e}_1]\!]\phi,\ \ldots,\ \mathsf{E}^\top[\![\mathbf{e_n}]\!]\phi\rangle$$

Our aim here, however, is to find needed arguments in internal calls and not just mappings of possible arguments to results. This requires a formalisation of the notion of *neededness*.

5.3 Argument needs

We will now consider a given program **prog** and assume that it may only be called from outside through the first function with a given set of arguments $\mathbf{V}_0 \subseteq \mathbf{D^k}$. Our aim is to describe the indirect calls to the functions in the program; that is all the possible calls to the functions which can be reached from this set of initial calls. We may formalise the notion of reachability as follows.

We define the function $\mathsf{A}[\![\mathbf{e}]\!]$ to return the set of argument tuples in calls to user-defined functions that will occur when evaluating the expression **e**. Possible calls are represented as closures of function numbers and arguments.

$$\langle \mathbf{i}, \nu\rangle \in \mathbf{C} = \{1,\ \ldots,\ \mathbf{n}\} \times \mathbf{D^k}$$

The function $\mathbf{A}[\![\mathbf{e}]\!]$ is defined as follows.

$$\mathbf{A}[\![\mathbf{e}]\!] : \Phi \to \mathbf{D^k} \to \mathcal{P}(\mathbf{C})$$

$$\begin{array}{lcl}
\mathbf{A}[\![\mathbf{x_i}]\!]\phi\nu & = & \varnothing \\
\mathbf{A}[\![\mathbf{c_i}]\!]\phi\nu & = & \varnothing \\
\mathbf{A}[\![\mathbf{op_i}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\nu & = & \mathbf{A}[\![\mathbf{e}_1]\!]\phi\nu \cup \cdots \cup \mathbf{A}[\![\mathbf{e_k}]\!]\phi\nu \\
\mathbf{A}[\![\mathbf{f_i}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\nu & = & \{\langle \mathbf{i},\, \mathbf{E}[\![\mathbf{e}_1]\!]\phi\nu,\ \ldots,\ \mathbf{E}[\![\mathbf{e_k}]\!]\phi\nu\rangle\} \cup \mathbf{A}[\![\mathbf{e}_1]\!]\phi\nu \cup \cdots \cup \mathbf{A}[\![\mathbf{e_k}]\!]\phi\nu
\end{array}$$

Using this function we may find the indirect calls to the functions in the program.

$$\mathcal{A}[\![\mathbf{prog}]\!]\phi\mathbf{S}_0 = \mathbf{fix}(\lambda\mathbf{S}.\ \mathbf{S}_0 \cup \bigcup_{\langle \mathbf{i},\, \nu\rangle \in \mathbf{S}} \mathbf{A}[\![\mathbf{e_i}]\!]\phi\nu)$$

The fixpoint is well-defined as the union operation is continuous. It is worth noting that the function $\mathbf{A}[\![\mathbf{e}]\!]$ is not necessarily continuous in either of its arguments. The function $\mathcal{A}[\![\mathbf{prog}]\!]$ is continuous in its second argument.

In the program

$$\mathbf{prog}\ =\ \mathbf{f}_1(\mathbf{x}_1,\ \ldots,\ \mathbf{x_k}) = \mathbf{e}_1,\ \ldots,\ \mathbf{f_n}(\mathbf{x}_1,\ \ldots,\ \mathbf{x_k}) = \mathbf{e_n}$$

with the initial call set $\mathbf{V}_0$ the indirect needs are $\mathcal{A}[\![\mathbf{prog}]\!](\mathbf{U}[\![\mathbf{prog}]\!])\{\langle 1, \nu\rangle \mid \nu \in \mathbf{V}_0\}$.

5.4 Propagation of abstract needs

The abstraction of argument needs is an interpretation which for each argument and each function assigns an abstract value in $\mathbf{D}^\top$. The abstract argument need interpretation is then defined as follows.

$$\mathbf{C}^\top = (\mathbf{D}^\top)^{\mathbf{k}}$$

$$\mathbf{A}^\top[\![\mathbf{e}]\!] : \Phi^\top \to (\mathbf{C}^\top)^{\mathbf{k}} \to (\mathbf{C}^\top)^{\mathbf{n}}$$

$$\begin{array}{lcl}
\mathbf{A}^\top[\![\mathbf{x_i}]\!]\phi\nu & = & \varnothing \\
\mathbf{A}^\top[\![\mathbf{c_i}]\!]\phi\nu & = & \varnothing \\
\mathbf{A}^\top[\![\mathbf{op_i}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\nu & = & \mathbf{A}^\top[\![\mathbf{e}_1]\!]\phi\nu \sqcup \cdots \sqcup \mathbf{A}^\top[\![\mathbf{e_k}]\!]\phi\nu \\
\mathbf{A}^\top[\![\mathbf{f_i}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\nu & = & \mathbf{only_i}(\mathbf{E}^\top[\![\mathbf{e}_1]\!]\phi\nu,\ \ldots,\ \mathbf{E}^\top[\![\mathbf{e_k}]\!]\phi\nu) \sqcup \mathbf{A}^\top[\![\mathbf{e}_1]\!]\phi\nu \sqcup \cdots \sqcup \mathbf{A}^\top[\![\mathbf{e_k}]\!]\phi\nu
\end{array}$$

Using this function we may find the indirect calls to the functions in the program.

$$\mathcal{A}^\top[\![\mathbf{prog}]\!]\phi\mathbf{S}_0 = \mathbf{fix}(\lambda\langle \mathbf{v}^1,\ \ldots,\ \mathbf{v^n}\rangle.\ \langle \mathbf{S}_0,\, \bot_{\mathbf{C}},\ \ldots\rangle \sqcup \mathbf{A}^\top[\![\mathbf{e}_1]\!]\phi\mathbf{v}^1 \sqcup \cdots \sqcup \mathbf{A}^\top[\![\mathbf{e_n}]\!]\phi\mathbf{v^n})$$

Example. Consider the following program

$$\begin{array}{ll} \mathbf{g}(\mathbf{x}) & = \mathbf{f}(1, \mathbf{x}) \\ \mathbf{f}(\mathbf{x}, \mathbf{y}) & = \mathbf{h}(\mathbf{x}, \mathbf{y} - 1) \\ \mathbf{h}(\mathbf{x}, \mathbf{y}) & = \mathbf{if}\ \mathbf{y} = 1\ \mathbf{then}\ \mathbf{x}\ \mathbf{else}\ \mathbf{h}(\mathbf{x} + 1, \mathbf{y} - 1) \end{array}$$

and let us assume that only the function **g** can be called externally. The initial description of abstract needs will be

$$\phi^0 = \langle\langle\top\rangle, \langle\bot, \bot\rangle, \langle\bot, \bot\rangle\rangle$$

The first iteration will register a call to **f**

$$\phi^1 = \langle\langle\top\rangle, \langle 1, \top\rangle, \langle\bot, \bot\rangle\rangle$$

The next iteration will see a call to **h**

$$\phi^2 = \langle\langle\top\rangle, \langle 1, \top\rangle, \langle 1, \top\rangle\rangle$$

From this description of argument we will reach the following new calls

$$\phi^{2'} = \langle\langle\top\rangle, \langle 1, \top\rangle, \langle 2, \top\rangle\rangle$$

which then should be lub'ed with ϕ^2 giving

$$\phi^3 = \langle\langle\top\rangle, \langle 1, \top\rangle, \langle\top, \top\rangle\rangle$$

With this the iteration stabilises and only the first argument to **f** has been recognised as constant.

6 Fixpoint iteration

In strictness analysis we may construct functions over the domain $\mathbb{2} = \{0, 1\}$ with $0 \leq 1$. We want to evaluate the function

$$\mathbf{f}^\sharp(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{y} \wedge \mathbf{z}) \vee \mathbf{f}^\sharp(\mathbf{z}, \mathbf{x}, \mathbf{f}^\sharp(\mathbf{y}, 1, 1))$$

for the arguments $\langle 0, 1, 1\rangle$, $\langle 1, 0, 1\rangle$, and $\langle 1, 1, 0\rangle$. Although the $\mathbf{f}^\sharp$ function looks like an ordinary function in a programming language, its implementation is not that simple. When we implement ordinary function we let the bottom value be represented as undefined or non-termination. This is not very attrictive when we evaluate the $\mathbf{f}^\sharp$ function since exactly the bottom element (0) is the interesting result which we would like to detect.

6.1 Fixpoint iteration

The safe way to evaluate a strictness function is to construct the ascending Kleene sequence, where we tabulate functions and iterate the function graph until stability. For the function

$$\mathbf{f}^\sharp(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{y} \wedge \mathbf{z}) \vee \mathbf{f}^\sharp(\mathbf{z}, \mathbf{x}, \mathbf{f}^\sharp(\mathbf{y}, 1, 1))$$

we start the fixpoint iteration with the bottom element: the constant function 0 and reevaluate the function for all possible arguments.

No \ Args	$\langle 0, 0, 0\rangle$	$\langle 0, 0, 1\rangle$	$\langle 0, 1, 0\rangle$	$\langle 0, 1, 1\rangle$	$\langle 1, 0, 0\rangle$	$\langle 1, 0, 1\rangle$	$\langle 1, 1, 0\rangle$	$\langle 1, 1, 1\rangle$
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	0	1	1	1	1	1
3	0	1	0	1	1	1	1	1
4	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1

In practice we are only interested in the function graph for some of the possible arguments. In this case we need the values of the following calls to detect the strictness.

$$\mathbf{f}(0, 1, 1) \quad \mathbf{f}(1, 0, 1) \quad \mathbf{f}(1, 1, 0)$$

When we evaluate these values we may need to evaluate the function for other arguments, so in general we will to tabulate the function for all possible arguments.

In some situations we may, however, restrict the iteration to a smaller set of arguments, and thus greatly simplify the computational task of finding the fixpoint. If we analyse the strictness of the function

$$\mathbf{h}(\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{v}, \mathbf{p}) = \mathbf{if}\ \mathbf{x} + \mathbf{y} = \mathbf{z} + \mathbf{v}\ \mathbf{then}\ \mathbf{p}\ \mathbf{else}\ \mathbf{h}(\mathbf{x} - 1, \mathbf{y} - 1, \mathbf{z} - 1, \mathbf{v} - 1, \mathbf{p} + 4)$$

then there are $2^5 = 32$ arguments in the function graph but we only need 5 arguments. In this section we will formalise the notion of needed arguments and show how it may be used to improve fixpoint iteration.

Language. As a start we will consider a simple language consisting of one recursively defined function over a complete partially ordered set $\mathbf{D}$. Programs $\mathbf{p}$ in this language will have the form

$$\mathbf{p} \quad : \quad \mathbf{letfix}\ \mathbf{f}(\mathbf{x}_1, \ \ldots, \mathbf{x_k}) = \mathbf{e_f}\ \mathbf{in}\ \mathbf{e}$$

where the expressions $\mathbf{e_f}$ and $\mathbf{e}$ are built from parameters, constants, basic operations, and function calls.

$$\begin{aligned}
\mathbf{e} : \ & \mathbf{x_i} \\
& \mid \mathbf{c_i} \\
& \mid \mathbf{op_i}(\mathbf{e}_1, \ \ldots, \mathbf{e_k}) \\
& \mid \mathbf{f}(\mathbf{e}_1, \ \ldots, \mathbf{e_k})
\end{aligned}$$

We may specify the semantics of this language using the functions M and E.

$$\begin{aligned}
&\mathsf{M}[\![\mathbf{letfix}\ \mathbf{f}(\mathbf{x}_1, \ \ldots, \mathbf{x_k}) = \mathbf{e_f}\ \mathbf{in}\ \mathbf{e}]\!] : \mathbf{D} \\
&\mathsf{E}[\![\mathbf{e}]\!] : (\mathbf{D^k} \to \mathbf{D}) \to \mathbf{D^k} \to \mathbf{D}
\end{aligned}$$

with

$$\begin{aligned}
&\mathsf{M}[\![\mathbf{letfix}\ \mathbf{f}(\mathbf{x}_1, \ \ldots, \mathbf{x_k}) = \mathbf{e_f}\ \mathbf{in}\ \mathbf{e}]\!] = \mathsf{E}[\![\mathbf{e}]\!](\mathbf{fix}\mathsf{E}[\![\mathbf{e_f}]\!])\bot_{\mathbf{D^k}} \\
&\mathsf{E}[\![\mathbf{x_i}]\!]\phi\rho = \mathbf{sel_i}(\rho) \\
&\mathsf{E}[\![\mathbf{c_i}]\!]\phi\rho = \underline{\mathbf{c}}_\mathbf{i} \\
&\mathsf{E}[\![\mathbf{op_i}(\mathbf{e}_1, \ \ldots, \mathbf{e_k})]\!]\phi\rho = \underline{\mathbf{op}}_\mathbf{i}(\mathsf{E}[\![\mathbf{e}_1]\!]\phi\rho, \ \ldots, \mathsf{E}[\![\mathbf{e_k}]\!]\phi\rho) \\
&\mathsf{E}[\![\mathbf{f}(\mathbf{e}_1, \ \ldots, \mathbf{e_k})]\!]\phi\rho = \phi(\mathsf{E}[\![\mathbf{e}_1]\!]\phi\rho, \ \ldots, \mathsf{E}[\![\mathbf{e_k}]\!]\phi\rho)
\end{aligned}$$

for constants $\underline{\mathbf{c_i}} \in \mathbf{D}$ and standard operations $\underline{\mathbf{op_i}} \in \mathbf{D^k} \to \mathbf{D}$. The function $\mathbf{sel_i}$ selects the $\mathbf{i}^{\text{th}}$ element in the tuple given as argument and "**fix**" finds the least fixpoint of its argument. To guarantee the well-definedness of this definition we must require that the function $\mathsf{E}[\![\mathbf{e_f}]\!]$ is continuous. This can be done by only using continuous standard operations $\underline{\mathbf{op_i}}$. It is, however, not necessary for all subexpressions of $\mathbf{e_f}$ to be continuous. Only the argument to "**fix**" needs to be continuous. In some applications (eg. data flow analysis) it is often natural locally to use non-monotonic operations while the fixpoint is found for a continuous function.

Fixpoint iteration. Consider the expression

$$\mathbf{letfix\ f}(\mathbf{x}_1,\ \ldots,\ \mathbf{x_k}) = \mathbf{e_f\ in\ f}(\mathbf{v}_1,\ \ldots,\ \mathbf{v_k})$$

with constants $\mathbf{v_i} \in \mathbf{D}$ and expression $\mathbf{e_f}$. The value of this expression is defined as

$$\mathbf{fix}(\mathsf{E}[\![\mathbf{e_f}]\!])\langle \mathbf{v}_1,\ \ldots,\ \mathbf{v_k}\rangle = \left(\bigsqcup_{\mathbf{i}} (\mathsf{E}[\![\mathbf{e_f}]\!])^{\mathbf{i}} \lambda\rho.\ \bot_{\mathbf{D}}\right) \langle \mathbf{v}_1,\ \ldots,\ \mathbf{v_k}\rangle$$

Our aim is to find an approximation to $\mathbf{fix}(\mathsf{E}[\![\mathbf{e_f}]\!])$ which has the correct value for $\langle \mathbf{v}_1,\ \ldots,\ \mathbf{v_k}\rangle$.

6.2 Argument needs

If we want to evaluate the expression $\mathbf{letfix\ f}(\mathbf{x}_1,\ \ldots,\ \mathbf{x_k}) = \mathbf{e_f\ in\ f}(\mathbf{v}_1,\ \ldots,\ \mathbf{v_k})$ we do not necessarily need to compute the fixpoint of $\mathbf{f}$ for all arguments in $\mathbf{D^k}$. Only arguments which may be reached from the call $\mathbf{f}(\mathbf{v}_1,\ \ldots,\ \mathbf{v_k})$ can influence the result. We may formalise the notion of reachability as follows.

We define the function $\mathsf{A}[\![\mathbf{e}]\!]$ to return the set of argument tuples in calls to the function $\mathbf{f}$ that will occur when evaluating the expression $\mathbf{e}$

$$\mathsf{A}[\![\mathbf{e}]\!] : (\mathbf{D^k} \to \mathbf{D}) \to \mathbf{D^k} \to \mathcal{P}(\mathbf{D^k})$$

$$\begin{aligned}
&\mathsf{A}[\![\mathbf{x_i}]\!]\phi\rho = \varnothing\\
&\mathsf{A}[\![\mathbf{c_i}]\!]\phi\rho = \varnothing\\
&\mathsf{A}[\![\mathbf{op_i}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\rho = \mathsf{A}[\![\mathbf{e}_1]\!]\phi\rho \cup \cdots \cup \mathsf{A}[\![\mathbf{e_k}]\!]\phi\rho\\
&\mathsf{A}[\![\mathbf{f}(\mathbf{e}_1,\ \ldots,\ \mathbf{e_k})]\!]\phi\rho = \{\langle \mathsf{E}[\![\mathbf{e}_1]\!]\phi\rho,\ \ldots,\ \mathsf{E}[\![\mathbf{e_k}]\!]\phi\rho\rangle\} \cup \mathsf{A}[\![\mathbf{e}_1]\!]\phi\rho \cup \cdots \cup \mathsf{A}[\![\mathbf{e_k}]\!]\phi\rho
\end{aligned}$$

It is worth noting that the function $\mathbf{A}[\![\mathbf{e}]\!]$ is not necessarily continuous in any of its arguments. This is because we have used a power set construction $\mathcal{P}(\mathbf{D^k})$ rather than a power domain and thereby forgetting the structure of $\mathbf{D}$. Continuity is, however, only important when we compute a fixpoint and in this case we may achieve continuity in a different way. Instead we define the function $\mathcal{A}$ as follows

$$\mathcal{A}[\![\mathbf{e}]\!]\phi\mathbf{S}_0 = \mathbf{fix}(\lambda\mathbf{S}.\ \mathbf{S}_0 \cup \bigcup_{\rho\in\mathbf{S}} \mathbf{A}[\![\mathbf{e}]\!]\phi\rho)$$

In the expression

$$\mathbf{letfix\ f}(\mathbf{x}_1,\ \ldots,\mathbf{x_k}) = \mathbf{e_f\ in\ f}(\mathbf{v}_1,\ \ldots,\mathbf{v_k})$$

the immediate argument need is $\rho^0 = \langle \mathbf{v}_1,\ \ldots,\mathbf{v_k}\rangle$. The indirect needs are $\mathcal{A}[\![\mathbf{e_f}]\!]\phi\{\rho^0\}$. This definition depends on the function environment $\phi = \mathbf{fix}(\mathbf{E}[\![\mathbf{e_f}]\!])$. The next step will be to show how argument needs may be used to simplify the fixpoint iteration. The definition is not directly useful since it depends on the fixpoint itself.

6.3 Iteration

For the expression

$$\mathbf{letfix\ f}(\mathbf{x}_1,\ \ldots,\mathbf{x_k}) = \mathbf{e_f\ in\ f}(\mathbf{v}_1,\ \ldots,\mathbf{v_k})$$

define the function

$$\mathbf{F_V}(\phi) = \lambda\rho.\ \mathbf{if}\ \rho\in\mathbf{V}\ \mathbf{then}\ \mathbf{E}[\![\mathbf{e_f}]\!]\phi\rho\ \mathbf{else}\ \phi(\rho)$$

which computes a better approximation to ϕ for arguments $\rho\in\mathbf{V}\subseteq\mathbf{D^k}$.

An *iteration sequence* is a sequence of function denotations ϕ^0, ϕ^1, ... and sets $\mathbf{V^i}\subseteq\mathbf{D}$ such that

$$\begin{aligned}\phi^0 &= \lambda\rho.\ \perp_{\mathbf{D}}\\ \phi^{\mathbf{i}+1} &= \mathbf{F_{V^i}}(\phi^{\mathbf{i}})\end{aligned}$$

An iteration sequence is then completely determined by the sets $\mathbf{V^i}$. The natural aim of fixpoint iteration is to keep these sets as small as possible while securing that the sequence of function denotations stabilise quickly

with the correct value for $\mathbf{f}(\mathbf{v_1}, \ldots, \mathbf{v_k})$. A simple way to achieve this is to use the function $\mathcal{A}$ to find the recursive needs from the initial needs.

$\mathbf{V} := \{\langle \mathbf{v_1}, \ldots, \mathbf{v_k} \rangle\}$
$\phi := \lambda\rho.\ \bot_{\mathbf{D}}$
repeat
  $\phi' := \phi$
  $\mathbf{V}' := \mathbf{V}$
  $\phi := \mathbf{F}_{\mathbf{V}}(\phi')$
  $\mathbf{V} := \mathcal{A}[\![\mathbf{e_f}]\!]\phi'\mathbf{V}'$
until $\phi = \phi' \wedge \mathbf{V} = \mathbf{V}'$

This algorithm will terminate if the domain $\mathbf{D}$ has finite height and the function environment ϕ will have the correct values of $\mathbf{fix}\mathsf{E}[\![\mathbf{e_f}]\!]$ for arguments in the final value of $\mathbf{V}$.

6.4 Improving the iteration

The algorithm presented above is quite simple but it may include arguments in the iteration which are not really needed. Such arguments are needed in early approximations but not needed in the fixpoint. We will here explain how that can happen and how they can be removed.

Example. Consider the function $\mathbf{f}$ over the powerset of $\{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}, \text{“}\mathbf{c}\text{”}\}$ with the subset ordering.

$$\mathbf{f}(\mathbf{d}) = \mathbf{f}(\mathbf{f}(\mathbf{f}(\mathbf{d} \cup \{\text{“}\mathbf{a}\text{”}\}) \cup \{\text{“}\mathbf{b}\text{”}\}) \cup \{\text{“}\mathbf{c}\text{”}\}) \cup \mathbf{d}$$

and an initial call $\mathbf{f}(\{\text{“}\mathbf{a}\text{”}\})$.

$\phi^0 = \lambda\mathbf{x}.\ \{\}$
$\mathbf{V}^0 = \{\{\text{“}\mathbf{a}\text{”}\}\}$
$\phi^1 = \phi^0[\{\text{“}\mathbf{a}\text{”}\} \mapsto \{\text{“}\mathbf{a}\text{”}\}]$
$\mathbf{V}^1 = \{\{\text{“}\mathbf{a}\text{”}\}, \{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}\}, \{\text{“}\mathbf{c}\text{”}\}\}$
$\phi^2 = \phi^1[\{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}\} \mapsto \{\text{“}\mathbf{a}\text{”}\}, \{\text{“}\mathbf{c}\text{”}\} \mapsto \{\text{“}\mathbf{a}\text{”}\}]$
$\vdots$
$\mathbf{V}^{\mathbf{s}} = \{\{\text{“}\mathbf{a}\text{”}\}, \{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}\}, \{\text{“}\mathbf{c}\text{”}\}, \{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{c}\text{”}\}, \{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}, \text{“}\mathbf{c}\text{”}\}\}$
$\phi^{\mathbf{s}} = \lambda\mathbf{s}.\ \{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}, \text{“}\mathbf{c}\text{”}\}$

With the fixpoint $\phi^{\mathbf{s}}$, however, we have

$$\mathbf{A}[\![\mathbf{e_f}]\!]\phi^{\mathbf{s}}\{\text{“}\mathbf{a}\text{”}\} = \{\{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}, \text{“}\mathbf{c}\text{”}\}\}$$
$$\mathbf{A}[\![\mathbf{e_f}]\!]\phi^{\mathbf{s}}\{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}, \text{“}\mathbf{c}\text{”}\} = \{\{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}, \text{“}\mathbf{c}\text{”}\}\}$$

So only the initial call $\mathbf{f}(\{\text{“}\mathbf{a}\text{”}\})$ and $\mathbf{f}(\{\text{“}\mathbf{a}\text{”}, \text{“}\mathbf{b}\text{”}, \text{“}\mathbf{c}\text{”}\})$ are really needed. The other calls are only needed in the approximations and iterating them to a fixpoint is not necessary.

Observation. The observation in the example is that we only need to evaluate the function for the needs that can be reached from the initial call. In other words (or symbols) we only need to compute approximations for arguments in $\mathcal{A}[\![\mathbf{e_f}]\!]\phi\{\rho^0\}$.

$$\begin{aligned}
\phi^0 &= \lambda\rho.\, \bot_{\mathbf{D}} \\
\mathbf{V}^0 &= \{\rho_0\} \\
\phi^1 &= \mathbf{F}_{\mathbf{V}^0}(\phi^{\mathbf{i}}) \\
\mathbf{V}^1 &= \mathcal{A}[\![\mathbf{e_f}]\!]\phi^1\mathbf{V}^0 \\
&\vdots \\
\phi^{\mathbf{i}+1} &= \mathbf{F}_{\mathbf{V}^{\mathbf{i}}}(\phi^{\mathbf{i}}) \\
\mathbf{V}^{\mathbf{i}+1} &= \mathcal{A}[\![\mathbf{e_f}]\!]\phi^{\mathbf{i}+1}\mathbf{V}^0
\end{aligned}$$

This may also be expressed as a small program

Fixpoint algorithm. We may compute the fixpoint efficiently using the following algorithm.

given $\mathbf{V}^0$
$\phi = \lambda\rho.\, \bot$
repeat
  $\mathbf{S} = \mathbf{V}^0$ {calls,which should be analysed}
  $\mathbf{V} = \varnothing$ {calls,which have been analysed}
  for $\rho \in \mathbf{S}$
    $\mathbf{V} = \mathbf{V} + \{\rho\}$
    $\mathbf{S} = \mathbf{S} + \mathbf{A}[\![\mathbf{e_f}]\!]\phi\rho - \mathbf{V}$
    $\phi = \phi[\rho \mapsto \mathbf{E}[\![\mathbf{e_f}]\!]\phi\rho]$
  end
until V and ϕ **stable**

An interesting property of this iteration strategy is that it will compute the correct fixpoint from possibly wrong (or imprecise) intermediate results. We know, however, that we approximate the solution from below so if we find a fixpoint, it will be the least.

In the algorithm we may use that the evaluations $\mathsf{A}[\![\mathbf{e_f}]\!]\phi\rho$ and $\mathsf{E}[\![\mathbf{e_f}]\!]\phi\rho$ can be done at the same time. The algorithm may be further optimised by using a depth-first strategy when collecting needed calls.

6.5 Example

Consider the strictness function

$$\mathbf{f}^\sharp(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{y} \wedge \mathbf{z}) \vee \mathbf{f}^\sharp(0, \mathbf{y}, \mathbf{f}^\sharp(\mathbf{x}, \mathbf{y}, \mathbf{z}))$$

with the call $\mathbf{f}^\sharp(0, 1, 1)$

$$\begin{array}{ll}
\phi^0 & = \lambda\rho.\ 0 \\
\mathbf{V}^0 & = \{\langle 0, 1, 1\rangle\} \\
\phi^1 & = \phi^0[\langle 0, 1, 1\rangle \mapsto 1] \\
\mathbf{V}^1 & = \{\langle 0, 1, 1\rangle, \langle 0, 1, 0\rangle\} \\
\phi^2 & = \phi^1[\langle 0, 1, 0\rangle \mapsto 0] \\
\mathbf{V}^2 & = \mathbf{V}^0 \\
\phi^3 & = \phi^2 \\
\mathbf{V}^3 & = \mathbf{V}^0
\end{array}$$

With the stabilised value only the initial call is needed but during the iteration also the call $\mathbf{f}(0, 1, 0)$ was used.

7 Bibliography

This section contains a bibliography of some important articles and books about abstract interpretation and domain theory. The list is far from complete but it can be used as a starting point for further search for references in these areas.

The majority of the articles concerning abstract interpretation where first published in the proceedings of a conference. These proceedings are sometimes published as a volume of journal. Especially Lecture Notes of Computer Science from Springer-Verlag is often used to publish such proceedings but also Sigplan Notices from ACM do publish proceedings. When we refer to a conference or a journal we often use an abbreviation. They are not always that easy to understand for people outside the area so to a start we list some often seen abbreviations.

Journals

Act Inf	Acta Informatica.
C.ACM	Communications of the ACM.
J.ACM	Journal of the ACM.
LNCS	Lecture Notes in Computer Science.
SCP	Science of Computer Programming.
Sigplan Not	ACM Sigplan Notices.
TCS	Theoretical Computer Science.
TOPLAS	ACM Transactions on Programming Languages and Systems.

Conferences

ESOP	European Symposium on Programming.
FPCA	Functional Programming and Computer Architecture.
ICALP	Int. Coll. on Automata, Languages and Programming.
LFP	Lisp and Functional Programming.
MFCS	Mathematical Foundation of Computer Science.
PLILP	Programming Language Implementation and Logic Programming.
POPL	Principles of Programming Languages.

The DIKU Library contain most of these journals and conference proceedings. Many of the references in this bibliography are to articles in LNCS, which has its own bookcase in the library. Some articles can be found in Sigplan Not. (see under S) and the POPL proceedings are published by ACM (see under A).

7.1 Books

There are no real text books on abstract interpretation but some books are useful as starting points for further search for references. An overview of some methods and techniques in abstract interpretation may be found in [1,7] and [9] is a good introduction to domain theory and denotational semantics.

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

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

[3] H P Barendregt. *The Lambda Calculus - Its Syntax and Semantics*. Volume 103 of Studies in logic and the foundation of mathematics, Revised edition. North-Holland, 1984.

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

[5] M Hecht. *Flow analysis of computer programs*. North-Holland, 1977.

[6] S T P Jones. *The implementation of functional programming languages*. Prentice-Hall, 1987.

[7] N D Jones and F Nielson. *Abstract Interpretation: a Semantics-Based Tool for Program Analysis*. Tech. Rep. DIKU, Univ. of Copenhagen, Denmark, 1989.

[8] H R Nielson and F Nielson. *Semantics with Applications: A formal Introduction*. John Wiley & Sons, 1992.

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

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

7.2 Abstract interpretation

Abstract interpretation started as an area with some articles by Patrick and Radhia Cousot in the late seventies. It is often [13] which is referenced as the founding article in the area. Abstract interpretation was introduced as a proof method for some program analyses method known as *data flow analysis* (see eg. [23,20]).

[11] F E Allen and J A Cocke. *A program data flow analysis procedure.* C. ACM **19**(3), pp. 137–147, 1976.

[12] P Cousot and R Cousot. *Static determination of dynamic properties of programs.* In 2nd International Symposium on Programming, Paris, France, 1976.

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

[14] P Cousot and R Cousot. *Automatic synthesis of optimal invariant assertions mathematical foundation.* In Symposium on Artificial Intelligence and Programming Languages, pp. 1–12. Volume 12(8) of ACM SIGPLAN Not., Aug., 1977.

[15] P Cousot and R Cousot. *Static determination of dynamic properties of generalised type unions.* In Conference on Language Design for Reliable Software, pp. 77–94. Volume 12(3) of ACM SIGPLAN Not., Mar., 1977.

[16] P Cousot. *Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotone sur un treilis, analyse sémantique des programmes.* Thése de Doctoratés Science Mathématiques. Univ. of Grenoble, France, 1978.

[17] P Cousot and R Cousot. *Static determination of dynamic properties of recursive procedures.* In Formal Description of Programming Concepts (E J Neuhold, ed.). North-Holland, 1978.

[18] P Cousot and R Cousot. *Systematic Design of Program Analysis Frameworks.* In 6th POPL, San Antonio, Texas, pp. 269–282, Jan., 1979.

[19] P Cousot. *Semantics Foundation of Program Analysis.* In Program Flow Analysis: Theory and Applications (S S Muchnick and N D Jones, eds.), chapter 10, pp. 303–342. Prentice-Hall, 1981.

[20] L D Fosdick and L J Osterweil. *Data Flow Analysis in Software Reliability.* Comp. Surv. **8**(4), pp. 305–330, Sept., 1976.

[21] N D Jones and S S Muchnick. *Flow analysis and optimization of Lisp-like structures.* In Program Flow Analysis: Theory and Applications (S S Muchnick and N D Jones, eds.), chapter 4, pp. 102–131. Prentice-Hall, 1981.

[22] N D Jones and S S Muchnick. *A flexible approach to interprocedural data flow analysis and programs with recursive data structures.* In 9th POPL, Albuquerque, NM, pp. 66–74, Jan., 1982.

[23] G Kildall. *A Unified Approach to Global Program Optimization.* In 1st POPL, pp. 194–206, Oct., 1973.

[24] T J Marlowe and B G Ryder. *Properties of data flow frameworks - a unified model.* Acta Inf. **28**(2), pp. 121–163, Dec., 1990.

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

[26] S S Muchnick and N D Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[27] P Naur. *Checking of Operand Types in Algol Compilers.* BIT **5**, pp. 151–163, 1965.

[28] J C Reynolds. *Automatic Computation of Data Set Definitions.* In IFIP'68, pp. 456–461, 1969.

7.3 Strictness analysis

The first work on strictness analysis is [52] which is an extract from [53]. The analysis has since been extended to higher-order functions [33] and lazy data structures [61].

[29] S Abramsky. *Strictness analysis and polymorphic invariance.* In Programs as Data Objects (H Ganzinger and N D Jones, eds.), pp. 1–23. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.

[30] S Abramsky and T P Jensen. *A relational approach to strictness for higher order polymorphic functions.* In 18th POPL, Orlando, Florida, pp. 49–54. ACM Press, Jan., 1991.

[31] A Bloss and P Hudak. *Variations on Strictness Analysis.* In LISP'86, Cambridge, Mass, pp. 132–142, Aug., 1986.

[32] G L Burn, C L Hankin, and S Abramsky. *Strictness analysis for higher-order functions.* Sci. Comp. Prog. **7**, pp. 249–278, 1986.

[33] G L Burn, C L Hankin, and S Abramsky. *The Theory of Strictness Analysis for Higher Order Functions.* In Programs as Data Objects (H Ganzinger and N D Jones, eds.), pp. 42–62. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.

[34] G L Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages.* Ph.D. Thesis. Imperial College, London, 1987.

[35] G L Burn. *Evaluation Transformers – A Model for the Parallel Evaluation of Functional Languages.* In FPCA'87, Portland, Oregon (G Kahn, ed.), pp. 446–470. Volume 274 of LNCS. Springer-Verlag, Sept., 1987.

[36] G L Burn. *A Relationship between Abstract Interpretation and Projection Analysis.* In 17th POPL, San Fransisco, California, pp. 151–155. ACM Press, Jan., 1990.

[37] G L Burn. *Using Projection Analysis in Compiling Lazy Functional Programs.* In LISP'90, Nice, France, pp. 227–241. ACM Press, 1990.

[38] P Dybjer. *Inverse Image Analysis.* In ICALP'87, Karlsruhe, Germany, pp. 21–30. Volume 267 of LNCS. Springer-Verlag, 1987.

[39] P Dybjer. *Inverse image analysis generalises strictness analysis.* Inform. and Comput. **90**(2), pp. 194–216, Feb., 1991.

[40] C V Hall and D S Wise. *Generating Function Versions with Rational Strictness Patterns.* Tech. Rep. Indiana Univ., 1988.

[41] P Hudak and J Young. *A set-theoretic characterisation of function strictness in the lambda calculus.* Tech. Rep. YALEU/DCS/RR-391. Yale Univ., 1985.

[42] P Hudak and J Young. *Higher-Order Strictness Analysis in Untyped Lambda Calculus.* In 13th POPL, St. Petersburg, Florida, pp. 97–109, Jan., 1986.

[43] J Hughes. *Strictness detection in non-flat domains.* In Programs as Data Objects (H Ganzinger and N D Jones, eds.), pp. 112–135. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.

[44] J Hughes. *Analysing strictness by abstract interpretation of continuations.* In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 4, pp. 63–102. Ellis-Horwood, 1987.

[45] J Hughes. *Backward Analysis of Functional Programs.* In Proceedings of the workshop on Partial Evaluation and Mixed Computation (D Bjørner et al., eds.), pp. 187–208. North-Holland, 1988.

[46] J Hughes. *Projections for Polymorphic Strictness Analysis.* In Category Theory and Comp. Sci., pp. 82–100. Volume 389 of LNCS. Springer-Verlag, 1989.

[47] S Hunt. *PERs generalise projections for strictness analysis.* In Proceedings of the Third Annual Glasgow Workshop on Functional Programming. Springer-Verlag, 1990.

[48] T P Jensen. *Strictness Analysis in Logical Form.* In FPCA'91, Cambridge, Mass, USA, pp. 352–366. Volume 523 of LNCS. Springer-Verlag, Aug., 1991.

[49] T M Kuo and P Mishra. *Strictness analysis: a new perspective based on type inference.* In FPCA'89, London, England, pp. 260–272. ACM Press, Sept., 1989.

[50] J Launchbury. *Projections for specialisation.* In Proceedings of the workshop on Partial Evaluation and Mixed Computation (D Bjørner et al., eds.), pp. 299–316. North-Holland, 1988.

[51] D Maurer. *Strictness Computation Using Special Lambda-Expressions.* In Programs as Data Objects (H Ganzinger and N D Jones, eds.), pp. 136–155. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.

[52] A Mycroft. *The Theory and Practice of Transforming Call-by-Need into Call-by-Value.* In International Symposium on Programming'80, Paris, France (B Robinet, ed.), pp. 269–281. Volume 83 of LNCS. Springer-Verlag, Apr., 1980.

[53] A Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. Thesis. Univ. of Edinburgh, Dec., 1981.

[54] A Mycroft. *Call-by-Need = Call-by-Value + Conditional.* Internal Report CSR-78-81. Univ. of Edinburgh, July, 1981.

[55] A Mycroft and M Rosendahl. *Minimal Function Graphs are not instrumented*. In WSA'92, Bordeaux, France, pp. 60–67. Bigre. Irisa Rennes, France, Sept., 1992.

[56] F Nielson. *Strictness Analysis and Denotational Abstract Interpretation*. Inform. and Comput. **76**, pp. 29–92, 1988.

[57] F Nielson and H R Nielson. *Finiteness conditions for strictness analysis*. In WSA'93, 1993.

[58] M Ogawa and S Ono. *Transformation of Strictness-Related Analyses based on Abstract Interpretation*. In International Conference on Fifth Generation Computer Systems, pp. 430–438, 1988.

[59] R C Sekar, S Pawagi, and I V Ramarkrishnan. *Small Domains Spell Fast Strictness Analysis*. In 17th POPL, San Fransisco, California, pp. 169–183. ACM Press, Jan., 1990.

[60] R C Sekar, P Mishra, and I V Ramakrishnan. *On the power and limitation of strictness analysis based on abstract interpretation*. In 18th POPL, Orlando, Florida, pp. 37–48. ACM Press, Jan., 1991.

[61] P Wadler and R J M Hughes. *Projections for Strictness Analysis*. In FPCA'87, Portland, Oregon (G Kahn, ed.), pp. 385–407. Volume 274 of LNCS. Springer-Verlag, Sept., 1987.

[62] P Wadler. *Strictness analysis on non-flat domains (by abstract interpretation)*. In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 12, pp. 266–275. Ellis-Horwood, 1987.

[63] P Wadler. *Strictness Analysis Aids Time Analysis*. In 15th POPL, San Diego, California. ACM Press, Jan., 1988.

[64] S Wray. *A new strictness detection algorithm*. In Workshop on Impl. of Func. Prog., Sweden, 1985.

7.4 Analysis of functional languages

Strictness analysis is normally an analysis of lazy functional languages. Other analysis problem for functional languages have also been expressed as abstract interpretations. They include compile time garbage collection [74], destructive uodates of arrays [73], globalising parameters [93] and complexity analysis [88].

[65] J M Barth. *Shifting garbage collection overhead to compile time.* C. ACM **20**(7), pp. 513–518, 1977.

[66] B Bjerner. *Time Complexity of Programs in Type Theory.* Ph.D. Thesis. Dept. of Comp. Sc., Univ. of Göteborg, Chalmers Univ. of Tech., 1989.

[67] B Bjerner and S Holmstrom. *A compositional approach to the time analysis of first order lazy functional languages.* In FPCA'89, London, England, pp. 157–165. ACM Press, Sept., 1989.

[68] A Bloss and P Hudak. *Path Semantics.* In Mathematical Foundation of Programming Language Semantics'87, New Orleans, Louisiana (M Mislove, ed.), pp. 476–489. Volume 298 of LNCS. Springer-Verlag, Apr., 1987.

[69] A Bloss. *Update analysis and the efficient implementation of functional aggregates.* In FPCA'89, London, England, pp. 26–38. ACM Press, Sept., 1989.

[70] O Danvy and A Filinsky. *Abstracting Control.* In LISP'90, Nice, France, pp. 151–160. ACM Press, 1990.

[71] B Goldberg. *Detecting sharing of partial applications in functional programs.* In FPCA'87, Portland, Oregon (G Kahn, ed.), pp. 408–425. Volume 274 of LNCS. Springer-Verlag, Sept., 1987.

[72] C K Gomard and P Sestoft. *Globalization and Live Variables.* Tech. Rep. DIKU, Univ. of Copenhagen, Denmark, 1990.

[73] P Hudak and A Bloss. *The aggregate update problem in functional programming systems.* In 12th POPL, New Oleans, Lousiana, pp. 300–314, Jan., 1985.

[74] P Hudak. *A semantic model of reference counting and its abstraction.* In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 3, pp. 45–62. Ellis-Horwood, 1987.

[75] P Hudak and J Young. *A Collecting Interpretation of Expressions (Without Powerdomains) -Preliminary Report-*. In 15th POPL, San Diego, California, pp. 107–118. ACM Press, Jan., 1988.

[76] J Hughes. *Abstract interpretation of first-order polymorphically typed languages*. In 1988 Glasgow Workshop on Functional Programming (C Hall, J Hughes, and J T O'Donnell, eds.), pp. 68–86. Volume 89/R4 of Research Report. Glasgow Univ., Feb., 1989.

[77] K Inoue, H Seki, and H Yagi. *Analysis of functional programs to detect run-time garbage cells*. ACM TOPLAS **10**(4), 1988.

[78] T Jensen and T Mogensen. *A backwards analysis for compile time garbage collection*. In ESOP'90, Copenhagen, Denmark, pp. 227–239. Volume 432 of LNCS. Springer-Verlag, 1990.

[79] N D Jones. *Flow analysis of lambda expressions*. In ICALP'81, pp. 114–128. Volume 115 of LNCS. Springer-Verlag, 1981.

[80] S Jones and D Metayer. *Compile-time garbage collection by sharing analysis*. In FPCA'89, London, England. ACM Press, Sept., 1989.

[81] P Jouvelot. *Semantic Parallelization: a practical execise in abstract interpretation*. In 14th POPL, Munich, West Germany, pp. 39–48, Jan., 1987.

[82] S Meira. *Optimised combinatoric code for applicative language implementation*. In International Symposium on Programming'84, Toulose, France. Volume 167 of LNCS. Springer-Verlag, Apr., 1984.

[83] M Odersky. *How to make destructive updates less destructive*. In 18th POPL, Orlando, Florida, pp. 25–36. ACM Press, Jan., 1991.

[84] S Ono, N Takahashi, and M Amamiya. *Optimized demand-driven evaluation of functional programs on a dataflow machine*. In ICPP, pp. 421–428. IEEE, 1986.

[85] S Ono. *Computation Path Analysis : Towards an Autonomous Global Dataflow Analysis*. In Proceedings of the 2nd France-Japan AI&CS Symposium, Sophia, France, 1987.

[86] U F Pleban and S S Muchnick. *A denotational semantics approach to program optimization*. Tech. Rep. Univ. of Kansas, 1980.

[87] P Roe. *A semantics for reasoning about parallel functional programs' performances.* Tech. Rep. Glasgow Univ., 1990.

[88] M Rosendahl. *Automatic Complexity Analysis.* In FPCA'89, London, England, pp. 144–156. ACM Press, Sept., 1989.

[89] D Sands. *Complexity analysis for a lazy higher-order language.* In ESOP'90, Copenhagen, Denmark, pp. 361–376. Volume 432 of LNCS. Springer-Verlag, 1990.

[90] D A Schmidt. *Detecting global variables in denotational definitions.* ACM TOPLAS **7**(2), pp. 299–310, 1985.

[91] J Schwarz. *Verifying the safe use of destructive operators in applicative programs.* In 3rd International Symposium on Programming, Paris, France, 1978.

[92] P Sestoft. *Replacing Function Parameters by Global Variables.* M.Sc. Thesis 88-7-2. DIKU, Univ. of Copenhagen, Denmark, Oct., 1988.

[93] P Sestoft. *Replacing function parameters by global variables.* In FPCA'89, London, England. ACM Press, Sept., 1989.

[94] O Shivers. *Control flow analysis in Scheme.* In SIGPLAN '88 Conference on PLDI, Atlanta, Georgia, pp. 164–174. Volume 23(7) of ACM SIGPLAN Not., July, 1988.

[95] B Steffen. *Optimal Run Time Optimization — Proved by a New Look at Abstract Interpretation.* In TAPSOFT'87, pp. 52–68. Volume 249 of LNCS. Springer-Verlag, 1987.

7.5 Analysis of logic programs

Analysis of logic programs using abstract interpretation is a relatively new and lively area. The earliest works considered the so-called "occur-check" problem in Prolog [119] but since then also anlyses detecting *freeness*, *sharing* and *groundness* have been constructed.

[96] R Barbuti, R Giacobazzi, and G Levi. *A declarative approach to abstract interpretation of logic programs.* Tech. Rep. TR-20/89. Univ. of Pisa, 1989.

[97] M Bruynooghe, B Demoen, A Callebaut, and G Janssens. *Abstract interpretation : Towards the global optimization of Prolog programs.* In IEEE Symposium on Logic Programming. IEEE, 1987.

[98] M Bruynooghe. *A Framework for the Abstract Interpretation of Logic Programs.* Tech. Rep. CW-62. Catholic Univ. of Leuven, Belgium, 1987.

[99] M Bruynooghe. *A practical framework for the abstract interpretation of logic programs.* J of Logic Programming **10**(2), pp. 91–124, Feb., 1991.

[100] S K Debray. *Dataflow analysis of logic programs.* Tech. Rep. Stony Brook, New York, 1986.

[101] S K Debray. *Global Optimization of Logic Programs.* Ph.D. Thesis. Stony Brook, New York, 1986.

[102] S K Debray and D S Warren. *Automatic mode inference for logic programs.* J of Logic Programming **5**(3), pp. 207–230, Sept., 1988.

[103] S K Debray. *Flow analysis of dynamic logic programs.* J of Logic Programming **7**(2), pp. 149–176, Sept., 1989.

[104] S K Debray. *Static Inference of Modes and Data Dependencies in Logic Programs.* ACM TOPLAS **11**(3), pp. 418–450. ACM, 1989.

[105] J Gallagher, M Codish, and E Shapiro. *Specialisation of Prolog and FCP Programs Using Abstract Interpretation.* In Proceedings of the workshop on Partial Evaluation and Mixed Computation (D Bjørner et al., eds.), pp. 159–186. Volume 6(2,3) of New Gener. Comput., 1988.

[106] J Gallagher, M Codish, and E Shapiro. *Specialisation of Prolog and FCP Programs Using Abstract Interpretation.* New Gener. Comput. **6**(2,3), pp. 159–186, 1988.

[107] C Hankin, D L Metayer, and D Sands. *Transformation of Gamma Programs.* In WSA'92, Bordeaux, France, pp. 12–19. Bigre. Irisa Rennes, France, Sept., 1992.

[108] K Horiuchi and T Kanamori. *Polymorphic type inference in Prolog by abstract interpretation.* In Logic Programming'87, pp. 107–116. Volume 315 of LNCS. Springer-Verlag, 1987.

[109] G Janssens and M Bruynooghe. *An instance of abstract interpretation integrating type and mode inferencing.* In International Conference on Logic Programming, pp. 669–683, 1988.

[110] N D Jones. *Concerning the abstract interpretation of prolog.* Tech. Rep. DIKU, Univ. of Copenhagen, Denmark, 1985.

[111] K Marriott and H Søndergaard. *Semantics-based dataflow analysis of logic programs.* In IFIP'89. North-Holland, 1989.

[112] K Marriott and H Søndergard. *Bottom-up Dataflow Analysis of Normal Logic Programs.* J of Logic Programming **13**(2-3), pp. 181–204, July, 1992.

[113] C S Mellish. *Some global optimisations for a Prolog compiler.* J of Logic Programming **2**(1), pp. 43–66, 1985.

[114] C S Mellish. *Abstract Interpretation on Prolog Programs.* In International Conference on Logic Programming, London, United Kingdom (E Shapiro, ed.), pp. 107–116. Volume 225 of LNCS. Springer-Verlag, July, 1986.

[115] C Mellish. *Abstract interpretation of prolog programs.* In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 8, pp. 181–198. Ellis-Horwood, 1987.

[116] U Nilsson. *Systematic construction of domains for abstract interpretation frameworks.* Tech. Rep. LITH-IDA-R-88-45. Linköping Univ., 1988.

[117] U Nilsson. *Towards a framework for the abstract interpretation of logic programs.* In PLILP'88, Orléans (P Deransart, B Lorho, and J Maluszynski, eds.), pp. 68–82. Volume 348 of LNCS. Springer-Verlag, May, 1988.

[118] D Plaisted. *The Occur-Check problem in Prolog.* In Proc. 1984 Symposium on Logic Programming, pp. 272–280, 1984.

[119] H Søndergaard. *An application of abstract interpretation of logic programs: occur-check reduction.* In ESOP'86, Saarbrucken, Germany, pp. 327–338. Volume 213 of LNCS. Springer-Verlag, 1986.

[120] H Søndergaard. *Semantics-Based Analysis and Transformation of Logic Programs.* Ph.D. Thesis 89/22. DIKU, Univ. of Copenhagen, Denmark, 1989.

[121] R Warren, M Hermenegildo, and S K Debray. *On the practicality of global flow analysis of logic programs.* In International Conference on Logic Programming, 1988.

7.6 Analysis of imperative languages

The earliest work on abstract interpretation [13] was based on a flow-diagram language. Since then most of the attentions has been centered around logic and functional languages. There are some works on analysis of imperative languages, mainly from the group around Cousot.

[122] F Bourdoncle. *Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity.* In PLILP'90, pp. 307–323. Volume 456 of LNCS. Springer-Verlag, 1990.

[123] P Cousot and N Halbwachs. *Automatic discovery of linear restraints among variables of a program.* In 5th POPL, Tuscon, AR, Jan., 1978.

[124] A Deutsch. *On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications.* In 17th POPL, San Fransisco, California, pp. 157–168. ACM Press, Jan., 1990.

[125] P Granger. *Static analysis of arithmetical congruences.* International Journal of Computer Mathematics, pp. 165–199, 1989.

[126] P Granger. *Static analysis of linear congruences among variables of a program.* Tech. Rep. LIX, Paris, France, 1990.

7.7 Analysis of other languages

Besides the more usual programming languages analyses of parallel languages and attribute grammars have also been expressed as abstract interpretations.

[127] H Christiansen. *Structure sharing in attribute grammars.* In PLILP'88, Orléans (P Deransart, B Lorho, and J Maluszynski, eds.), pp. 180–200. Volume 348 of LNCS. Springer-Verlag, May, 1988.

[128] P Cousot and R Cousot. *Semantic analysis of communicating sequential processes.* In ICALP'80. Volume 85 of LNCS. Springer-Verlag, 1980.

[129] F Nielson. *Towards Viewing Nondeterminism as Abstract Interpretation.* Foundations of Software Technology & Theor. Comp. Sci. **3**, 1983.

[130] J H Reif. *Data flow analysis of distributed communicating processes.* Tech. Rep. TR-12-83. Harward Univ., Sept., 1984. Also in POPL6.

[131] M Rosendahl. *Strictness analysis for attribute grammars.* In PLILP'92, pp. 145–157. Volume 631 of LNCS. Springer-Verlag, 1992.

7.8 Fundamental studies

A number of works have proposed other frameworks or techniques for specification and proofs in abstract interpretation. This includes the use of inductive relations [142,132], minimal function graphs [137] and analyses of functional [52] and logic [120] languages.

[132] S Abramsky. *Abstract interpretation, logical relations and Kan extensions.* Journal of logic and computation **1**(1), 1990.

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

[134] V Donzeau-Gouge. *Denotational definitions of properties of program computations.* In Program Flow Analysis: Theory and Applications (S S Muchnick and N D Jones, eds.), chapter 11, pp. 343–379. Prentice-Hall, 1981.

[135] J Hughes and J Launchbury. *Reversing Abstract Interpretations.* In ESOP'92, pp. 269–286. Volume 582 of LNCS. Springer-Verlag, 1992.

[136] N D Jones and S S Muchnick. *Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra.* In Program Flow Analysis: Theory and Applications (S S Muchnick and N D Jones, eds.), chapter 12, pp. 380–393. Prentice-Hall, 1981.

[137] N D Jones and A Mycroft. *Data Flow Analysis of Applicative Programs using Minimal Function Graphs.* In 13th POPL, St. Petersburg, Florida, pp. 296–306, Jan., 1986.

[138] N D Jones. *Flow Analysis of Lazy Higher Order Functional Programs.* In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 5, pp. 103–122. Ellis-Horwood, 1987.

[139] N D Jones and H Søndergaard. *A semantics-based framework for abstract interpretation of continuations.* In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 6, pp. 123–142. Ellis-Horwood, 1987.

[140] R B Kieburtz and M Napierala. *Abstract Semantics.* In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 7, pp. 143–180. Ellis-Horwood, 1987.

[141] A Mycroft and F Nielson. *Strong Abstract Interpretation using Power Domains*. In ICALP'83, Barcelona, Spain, pp. 536–547. Volume 154 of LNCS. Springer-Verlag, July, 1983.

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

[143] A Mycroft and M Rosendahl. *Minimal Function Graphs are not instrumented*. In WSA'92, Bordeaux, France, pp. 60–67. Bigre. Irisa Rennes, France, Sept., 1992.

[144] P Panangaden and P Mishra. *A category theoretic formalism for abstract interpretation*. Tech. Rep. UUCS-84-005. Utah Univ., 1984.

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

[146] M Rosendahl. *Abstract Interpretation using Attribute Grammars*. In WAGA'90 (P Deransart and M Jourdan, eds.), pp. 143–156. Volume 461 of LNCS. Springer-Verlag, Oct., 1990.

7.9 Domain theory

Domain theory is a special branch of lattice theory which have G Birkhoff [147] as a central figure. The application to semantics and many of the results related to recursively defined domains are due to Dana Scott [161,160].

[147] G Birkhoff. *Lattice Theory*, third edition. American Mathematical Society, 1973.

[148] P Cousot and R Cousot. *Abstract Interpretation and applications to logic programs*. J of Logic Programming **13**(2-3), pp. 103–180, July, 1992.

[149] J Gallagher and M Bruyhooghe. *The Derivation of an Algorithm for Program Specialisation*. New Gener. Comput. **9**, pp. 305–333, 1991.

[150] J A Goguen, J W Thatcher, E G Wagner, and J B Wright. *Initial algebra semantics and continuous algebras*. J. ACM. **24**(1), 1977.

[151] C Hankin and S Hunt. *Approximate Fixed Points in Abstract Interpretation*. In ESOP'92, pp. 219–232. Volume 582 of LNCS. Springer-Verlag, 1992.

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

[153] R Milne and C Strachey. *A theory of programming language semantics.* Chapman and Hall, 1976.

[154] F Nielson. *Tensor products generalize the relational data flow analysis method.* In Proc 4th Hungarian Comp Sci Conf, pp. 211–225, 1985.

[155] F Nielson and H R Nielson. *Finiteness Conditions for Fixed Point Iteration.* In LISP'92, San Francisco, CA, pp. 96–108. ACM Press, 1992.

[156] G D Plotkin. *A powerdomain construction.* SIAM J. Comp. **5**, pp. 452–487, 1976.

[157] G D Plotkin. *A Structural Approach to Operational Semantics.* Tech. Rep. FN-19. DAIMI, Univ. of Aarhus, Denmark, Sept., 1981.

[158] J C Reynolds. *On the relation between direct and continuation semantics.* In ICALP'74, pp. 141–156. Volume 14 of LNCS. Springer-Verlag, 1974.

[159] D Scott. *The Lattice of Flow Diagrams.* Tech. Rep. PRG-3. Oxford Univ., Nov., 1970.

[160] D Scott. *Data Types as Lattices.* SIAM J. Comp. **5**(3), pp. 522–587, Sept., 1976.

[161] D S Scott. *Domains for Denotational Semantics.* In ICALP'82, Aarhus, Denmark, pp. 577–613. Volume 140 of LNCS. Springer-Verlag, 1982.

[162] M B Smyth. *Power Domains.* J. Comput. System Sci. **16**, pp. 23–36, 1978.

[163] M B Smyth and G Plotkin. *The category-theoretic solution of recursive domain equations.* SIAM J. Comp. **11**(4), pp. 761–783, Nov., 1982.

[164] A Tarski. *A lattice-theoretical fixpoint theorem and its applications.* Pacific J. Math. **5**, pp. 285–309, 1955.

7.10 Binding time analysis

Binding time analysis is essentially a Copenhagen speciality. The analysis is a central part of a partial evaluator.

[165] L O Andersen and C Mossin. *Binding Time Analysis via Type Inference*. Student Project 90-10-12. DIKU, Univ. of Copenhagen, Denmark, Oct., 1990.

[166] A Bondorf, N D Jones, T Mogensen, and P Sestoft. *Binding time analysis and the taming of self-application*. Tech. Rep. DIKU, Univ. of Copenhagen, Denmark, 1988.

[167] A Bondorf. *Self-Applicable Partial Evaluation*. Ph.D. Thesis 90/17. DIKU, Univ. of Copenhagen, Denmark, 1990.

[168] C Consel. *Binding Time Analysis for Higher Order Untyped Functional Languages*. In LISP'90, Nice, France, pp. 264–272. ACM Press, 1990.

[169] C Consel and O Danvy. *From Interpreting to Compiling Binding Times*. In ESOP'90, Copenhagen, Denmark, pp. 88–105. Volume 432 of LNCS. Springer-Verlag, 1990.

[170] C Consel and O Danvy. *Static and Dynamic Semantic Processing*. In 18th POPL, Orlando, Florida, pp. 14–24. ACM Press, Jan., 1991.

[171] C K Gomard and N D Jones. *A partial evaluator for the untyped lambda-calculus*. J of Functional Programming **1**(1), pp. 21–69, Jan., 1991.

[172] G W Hamilton. *Sharing Analysis of Lazy First-Order Functional Programs*. In WSA'92, Bordeaux, France, pp. 68–78. Bigre. Irisa Rennes, France, Sept., 1992.

[173] N D Jones and S S Muchnick. *Automatic optimization of binding times*. In 3rd POPL, Atlanta, Georgia, pp. 77–94, Jan., 1976.

[174] N D Jones and S S Muchnick. *Binding Time Optimization in Programming Languages: Some Thoughts Toward the Design of an Ideal Language*. In 3rd POPL, Atlanta, Georgia, pp. 77–94, Jan., 1976.

[175] N D Jones and S S Muchnick. *TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages*. Volume 66 of LNCS. Springer-Verlag, 1978.

[176] N D Jones, P Sestoft, and H Søndergaard. *An Experiment in Partial Evaluation: The Generation of a Compiler Generator.* In Rewriting Techniques and Applications, pp. 124–140. Volume 202 of LNCS. Springer-Verlag, 1985.

[177] N D Jones. *Static Semantics and Binding Time Analysis.* Working paper. DIKU, Univ. of Copenhagen, Denmark, 1988.

[178] N D Jones, P Sestoft, and H Søndergaard. *Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation.* Lisp and Symbolic Computation **2**(1), pp. 9–50, 1989.

[179] J Launchbury. *Dependent Sums Express Separation of Binding Times.* In 1989 Glasgow Workshop on Functional Programming, Glasgow, Scotland (K Davis and J Hughes, eds.), pp. 238–253. Springer-Verlag, 1990.

[180] T Mogensen. *Partially static structures in a self-applicable partial evaluator.* In Proceedings of the workshop on Partial Evaluation and Mixed Computation (D Bjørner et al., eds.), pp. 325–347. North-Holland, 1988.

[181] T Mogensen. *Binding Time Analysis for Polymorphically Typed Higher Order Languages.* In TAPSOFT'89, Barcelona, Spain (J Diaz and F Orejas, eds.), pp. 298–312. Volume 352 of LNCS. Springer-Verlag, Mar., 1989.

[182] T Mogensen. *Binding Time Aspects of Partial Evaluation.* Ph.D. Thesis. DIKU, Univ. of Copenhagen, Denmark, 1989.

[183] T Mogensen. *Separating Binding Times in Language Specifications.* In FPCA'89, London, England, pp. 14–25. ACM Press, Sept., 1989.

[184] H R Nielson and F Nielson. *Automatic Binding Time Analysis for a Typed λ-calculus.* Sci. Comp. Prog. **10**(2), pp. 139–176, 1988.

[185] S A Romanenko. *Arity Raiser and Its Use in Program Specialization.* In ESOP'90, Copenhagen, Denmark, pp. 341–360. Volume 432 of LNCS. Springer-Verlag, 1990.

[186] W L Scherlis. *Program Improvement by Internal Specialization.* In 8th POPL, Williamsburg, Va, pp. 41–49, Jan., 1981.

[187] W Winsborough. *Multiple Specialization using Minimal-Function Graph Semantics.* J of Logic Programming **13**(2-3), pp. 259–290, July, 1992.

7.11 Analysis of the semantic metalanguage

The idea of making abstract interpretations of the semantic metalanguage used in denotational semantics have been engineered by Flemming and Hanne Nielson from Århus University. This is often referred to as *denotational abstract interpretation*. An introduction to this area may be found in [7].

[188] A Mycroft. *A Study on Abstract Interpretation and "Validating Microcode Algebraically"*. In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 9, pp. 199–218. Ellis-Horwood, 1987.

[189] F Nielson. *A Denotational Framework for Data Flow Analysis*. Acta Inf. **18**, pp. 265–287, 1982.

[190] F Nielson. *Abstract Interpretation using Domain Theory*. Ph.D. Thesis. Univ. of Edinburgh, Oct., 1984.

[191] F Nielson. *Program Transformation in a Denotational Setting*. ACM TOPLAS **7**(3), pp. 359–379, July, 1985.

[192] F Nielson. *Abstract Interpretation of Denotational Definitions*. In STACS'86, pp. 1–20. Volume 210 of LNCS. Springer-Verlag, 1986.

[193] F Nielson. *Expected Forms of Data Flow Analysis*. In Programs as Data Objects (H Ganzinger and N D Jones, eds.), pp. 172–191. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.

[194] H R Nielson and F Nielson. *Pragmatic aspects of two-level denotational meta-languages*. In ESOP'86, Saarbrucken, Germany. Volume 213 of LNCS. Springer-Verlag, 1986.

[195] H R Nielson and F Nielson. *Semantics Directed Compiling for Functional Languages*. In LISP'86, Cambridge, Mass, pp. 249–257, Aug., 1986.

[196] Nielson and Nielson. *A tutorial on TML: the metalanguage of the PSI-project*. Tech. Rep. 86-4. Aalbog Universitetscenter, May, 1986.

[197] F Nielson. *Towards a Denotational Theory of Abstract Interpretation*. In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 10, pp. 219–245. Ellis-Horwood, 1987.

[198] H R Nielson. *The core of the PSI-system (version 1.0).* Tech. Rep. IR-87-02. Aalbog Universitetscenter, Mar., 1987.

[199] F Nielson and H R Nielson. *The TML-approach to compiler-compilers.* Tech. Rep. ID-TR-193988-47. The Technical Univ. of Denmark, 1988.

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

[201] F Nielson. *Two-level semantics and abstract interpretation.* Theor. Comp. Sci. **69**, pp. 117–242, 1989.

7.12 Implementation of fixpoint iteration

Implementations of abstract interpretations are normally based on fixpoint iteration. There are a number of algorithms which improves on a simple breadth-first iteration by using properties of the domains or the equations. Especially the so-called *frontier*-algorithm for strictness analysis has attracted some attention.

[202] C Clack and S P Jones. *Strictness analysis—a practical approach.* In FPCA'85, Nancy, France, pp. 35–49. Volume 201 of LNCS. Springer-Verlag, Sept., 1985.

[203] A Ferguson and J Hughes. *Fast abstract interpretation using sequential algorithms.* In WSA'93, 1993.

[204] C Hankin and D L Metayer. *Deriving Algorithms From Type Inference Systems: Application to Strictness Analysis.* In POPL'94, 1994.

[205] C Hankin and D L Metayer. *Lazy type inference for the strictness analysis of lists.* In ESOP'94, 1994.

[206] C Hankin and D L Metayer. *A Type-based Framework for Program Analysis.* In SAS'94, 1994.

[207] S Hunt. *Frontiers and open sets in abstract interpretation.* In FPCA'89, London, England, pp. 1–13. ACM Press, Sept., 1989.

[208] K D Jensen, P Hjaeresen, and M Rosendahl. *Efficient Strictness Analysis of Haskell.* In SAS'94, 1994.

[209] S P Jones and C Clack. *Finding fixpoints in abstract interpretation.* In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 11, pp. 246–265. Ellis-Horwood, 1987.

[210] Q Ma and J C Reynolds. *Types, Abstraction, and Parametric Polymorphism (part 2).* In Mathematical Foundation of Programming Semantics'91 (M Main et al., eds.), pp. 1–40. Volume 598 of LNCS. Springer-Verlag, Mar., 1991.

[211] C Martin and C Hankin. *Finding Fixed Points in Finite Lattices.* In FPCA'87, Portland, Oregon (G Kahn, ed.), pp. 426–445. Volume 274 of LNCS. Springer-Verlag, Sept., 1987.

[212] L Mauborgne. *Abstract Interpretation using TDGs.* In SAS'94, 1994.

[213] M
Rosendahl. *Higher-Order Chaotic Iteration Sequences.* In PLILP'93, pp. 332–345. Volume 714 of LNCS. Springer-Verlag, 1993.

[214] J Young and P Hudak. *Finding fixpoints on functional spaces.* Tech. Rep. YALEEU/DCS/RR-505. Yale Univ., Dec., 1986.

Contents