

Automatic Program Analysis

Mads Rosendahl
Institute of Datalogy
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø
Denmark

Summary

The objective of this report is to show how *time bound functions* can be constructed automatically in a formal semantics foundation. Such functions binding the time requirements of algorithms are in the size of their input. The construction method developed here will always terminate. In accordance with the undecidability of the halting problem the resulting time bound functions are in general not total. However it has turned out to give efficient results for many programs.

The construction algorithm is based on abstract interpretation and the generated time bound functions are shown to bound the time requirements. The algorithm consists in three phases: first part generates a time bound program that computes a safe approximation of the time bound function in a specially constructed semantics; next part compiles the program to the standard semantics with the use of information gained in a data flow analysis; finally the time bound program is optimized in a transformation system, which also solves finite-difference equations.

The system is implemented in Lisp. A number of programs have been automatically analyzed and the results are presented.

Current Address: University of Cambridge, Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, England.

This report has been submitted as a Master's Thesis at University of Copenhagen (Report nr. 86-12-1).

Contents

1	Introduction	3
2	Overview	7
3	Semantics of Step Counting Versions	11
3.1	Language.	11
3.2	Semantics.	12
3.3	Step Counting Version	13
3.4	A Semantic Framework.	15
4	Time Bound Programs	19
4.1	Collecting Semantics	22
4.2	The Domain of partly known structures.	27
4.3	Extending Functions	31
4.4	The \tilde{V}_\perp -Semantics.	37
4.5	Termination (nontermination).	39
4.6	Inverted length functions.	39
4.7	Program Composition.	41
4.8	Compilation	42
5	Data Flow Analysis	46
5.1	Collecting Semantics	48
5.2	The Domain of Sets of Partly Known Structures	54
5.3	Term versions	56
5.4	The Abstract Interpretation.	64
5.5	Correctness.	66
5.6	Compilation.	71
5.7	Status so far.	73
6	A Concept of Driving	75
6.1	6.1 Example	76
6.2	Basic Configurations	77
6.3	Driving algorithm	79
6.4	Status	81

7	Finite-Difference Equations	85
7.1	Classes of finite-difference equations.	85
7.2	Simple decrement.	86
7.3	General decrement.	89
8	Results	91
8.1	Results.	91
8.2	Description of the system	96
9	Conclusion	98
10	Bibliography	100
A	Output from the analyser system	105
A.1	Lookup	106
A.2	Union	107
A.3	Reverse	108
A.4	Parse	109
A.5	Sort	115

Chapter 1

Introduction

Time bound functions for programs are well known concepts in complexity theory (e.g. [Papadimitriou,82]), but the literature contains only few papers on how to construct the time bound functions automatically. The main attack is probably still the one done by Ben Wegbreit in his "Mechanical Program Analysis" paper from 1975. He describes the performance of programs in a first order Lisp subset by four closed-form expressions, all functions in the length of the arguments. This four tuple express the minimal and the maximal running time, plus the mean and the variance of the running time. His work reveals many obstacles in the area, even though he is only able to handle fairly simple programs.

A survey of computer-assisted program analysis is given in [Cohen,82], and besides two different systems are described in [Metayer,85] and [Kasai,80]. There is a tendency in these works to defeat the restrictions of the Wegbreit work by either using heuristics, for instance by annotation. The heuristics are used for two reasons:

1. A time bound function express a computational property (the running time) of a program from the size of input, and the sets of input with the same size will in general be infinite.
2. To construct closed-form expressions requires highly optimizing steps, eg. solution of difference equations.

The objective of this paper is to develop a formally founded construction algorithm. The method doesn't avoid a sort of annotation, but it is a simple and fairly obvious one. All what is needed is a precise definition of what is meant by "size of input". The right meaning may change from program to program and it depends on how input is used. In the work of Wegbreit heuristics is used to decide whether to use length of list or number of cons cells as the right size measure, but it was not evident from the result what was chosen.

The algorithm here is based on abstract interpretation [Cousot & Cousot, 77], and the language is nearly the same as in [Wegbreit, 1975]. We show this type of program analysis can be formalized and reduced to a program transformation/partial evaluation problem for a large class of programs. In the abstract

interpretation framework the construction method is proven correct as the time bound functions if defined will give a safe bound on the time requirements.

Here we can only give an intuitive idea of the method described precisely later. We use a semantics function \mathbf{L} to give program meaning:

1. Given a program p with

$$\mathbf{L} p : Data \rightarrow Data.$$

Construct a program t_p to compute the running time of p

$$\mathbf{L} t_p : Data \rightarrow Time$$

2. Generalize t_p to t_p' with powersets

$$\mathbf{L} t_p' : \mathbf{P}(Data) \rightarrow \mathbf{P}(Time)$$

3. Let \max give the maximum of a set of numbers

$$\mathbf{L} \max : \mathbf{P}(Time) \rightarrow Time$$

4. Given size as a definition of “size of input”

$$\mathbf{L} \text{size} : Data \rightarrow Numbers$$

Construct size^{-1} with

$$\mathbf{L} \text{size}^{-1} : Numbers \rightarrow \mathbf{P}(Data)$$

5. Finally

$$\mathbf{L} (\max \circ t_p' \circ \text{size}^{-1}) : Numbers \rightarrow Time$$

In the first part we give this intuitive idea a meaning by showing how it is possible to construct a program (time bound program) that in a specially constructed semantics computes a safe approximation to the time bound function. In a second part the program is compiled to the standard semantics. The compilation uses information about possible values of expressions found by a data flow analysis inspired by [Reynolds, 1968] Finally the time bound program in the standard semantics is optimized by a simple transformation system using the principle of driving [Turchin, 86] as is done in [Wadler, 84]. The transformation system can solve a number of difference equations.

Applications.

A main reason for this project was to use time bound functions in a program transformer. Cost information can be used to make good choices between possible program parts. Program transformation systems (eg. [Burstall & Darlington, 1977]) normally contains a number of possible transformation rules, and the system makes more or less indeterminate choices between these rules. Algorithms to select the rule resulting in the most efficient residual program could be used to make some of the choices deterministic, or at least less arbitrary.

Transforming sequential programs to parallel programs depends heavily on deciding what can be computed effectively in parallel. Recently Sarkar & Hennessy have published a paper ([Sarkar & Hennessy, 1986]) where they develop a system to schedule tasks to processors in a parallel program. Their system uses profile data to estimate running times of tasks. It could be interesting to see how far one can go in making the transformation automatic.

Time Bound Function and the Halting Problem.

A time bound function is a *partial* function that gives an upper bound of the running time of a program from the size of input. The maximal running time is in general an undecidable property of programs, and it will be difficult (and probably not interesting) to define a class of programs where the maximal running time is decidable. It will be too strong to require that the time bound function should give the maximal running time over the input set in mention. A relaxation is just to give an upper bound of running time. This means that even if the time bound function is undefined the program could terminate for all input of the given size.

It is not of decisive importance to allow the time bound function to be partial. If the function should be total the domain must contain an infinity symbol denoting that the program might diverge. As the time bound function is computed by a program (called a time bound program) a watch could be incorporated in this program and if the time bound wasn't found in let us say two hours then it should output the infinity symbol.

Limitations and Results.

The algorithm produces a time bound program, and as it is a syntactical object there are two degrees of success for an analyzer algorithm.

1. The time bound program terminates.
2. The time bound program is in closed form.

The first degree of success is obtained for a large class of programs where termination is secured without an "intelligent" proof. The class excludes as example

programs that compute fixpoints, and of course also any general program interpreter. It has been used successfully on a suite of programs containing sorting algorithms, matrix algorithms and a deterministic parser resulting in total and well approximating time bound functions

The second degree of success concerns the efficiency (or computation time of the time bound program!) and this is obtained for a smaller class of programs depending on the optimizations implemented. This system solves a variety of difference equations, but the class of programs can be extended with a program transformation system using many heuristical rules. As the purpose of this work has been to make a semantical based deterministic analyzer only general rules are used. The execution times of the generated time bound programs are typically of the same size or smaller than running the program with arbitrarily chosen input.

About the report.

The algorithm is developed in a formal semantics framework, and the reader is assumed to be familiar with abstract interpretation and flow analysis. (eg. [Cousot & Cousot, 1977] and [Jones & Mycroft, 1986]). Some knowledge of the works [Wegbreit 1975] and [Reynolds 1969] will be an advantage.

The following two definitions are assumed to be well-known:

Complete lattice. Partially ordered set where any nonempty subset have least upper bound and greatest lower bound (see [Cousot & Cousot, 1977]).

Domain (cpo). Partially ordered set with least element, where any (ordered) chain of elements have a least upper bound (see [Bird 1976]).

The proof of the algorithm is given in chapter 4 and 5 and especially chapter 5 requires both good foundation and patience of the reader. In chapter 2 we give an introductory example and chapter 3 describes the semantics framework to be used in the rest of the work including to give the semantics of the language. Chapter 6 and 7 present a transformation system to improve time bound programs. Chapter 8 discusses the result from an implementation of the algorithm, and finally it is compared with other works in the area.

Chapter 2

Overview

Given a program p . A *time bound program* for p is a program \mathbf{tb}_p that from input n_1, \dots, n_k computes an upper bound of the running time for p with inputs of size n_1, \dots, n_k .

To formalize the definition we use a semantics function \mathbf{L} to give program text a meaning:

$$\mathbf{L} : \text{program} \rightarrow V_{\perp}^k \rightarrow V_{\perp}$$

The function is total, and the set V_{\perp} is S-expressions of Lisp extended with an undefined element \perp to denote nontermination. The semantics is described in chapter 3, and it is shown how to construct a program \mathbf{t}_p to compute the running time of p from the arguments.

$$\mathbf{L} \mathbf{t}_p \langle x_1, \dots, x_k \rangle = \text{running time of } p \text{ with input } \langle x_1, \dots, x_k \rangle$$

The set of atoms in S-expressions must include numbers to represent the running times.

In the paper we develop an algorithm to construct a program \mathbf{tb}_p such that

$$\mathbf{L} \mathbf{tb}_p \langle n_1, \dots, n_k \rangle \geq \max\{\mathbf{L} \mathbf{t}_p \langle x_1, \dots, x_k \rangle \mid \forall x_i : \text{length}(x_i) = n_i\}$$

where \geq is a partial correctness ordering : If the time bound program terminates then will the right hand side be defined and the left hand side will be greater or equal to the right hand side. The function

$$\text{length} : V \rightarrow \text{Numbers}$$

gives the conceptsize of input a meaning, but as the right meaning may change from program to program it must in some way be supplied by the user. How this can be done will be explained in more details in chapter 4.

Example.

In the rest of this chapter we give a simple example to illustrate the ideas in generating time bound programs. The program will be the `union` function from Lisp. It is simple but not trivial since it was not possible for the Wegbreit algorithm to analyze the program. This definition is taken from [Wegbreit, 1975] :

```

union(x,y) = (if (eq x nil) then y
              else
              (if (call member (car x) y)
                  then (call union (cdr x) y)
                  else (cons (car x) (call union (cdr x) y))
              ))

```

```

member(x,y) = (if (eq y nil) then false
                else
                (if (eq x (car y)) then true
                    else (call member x (cdr y))
                ))

```

Step counting version. The first step is to construct a program to compute the running time of `union`. Instead of computing the actual machine dependent execution time we choose to count the number of basic operations performed under the evaluation. We count "1" for fetching a parameter or constant, for a basic operation (built-in function) we count "1" plus the time to evaluate the arguments. For function calls it is the time to evaluate the body plus the time to evaluate the arguments.

This new program will be called `t-union` and we call such programs for *step counting versions*. The construction algorithm is formalized in chapter 3.

```

t-union(x,y) = (if (eq x nil) then 4
                 else
                 (if (call member (car x) y)
                     then (add 11
                             (add (call t-member (car x) y)
                                   (call t-union (cdr x) y))
                             )
                     else (add 14
                             (add (call t-member (car x) y)
                                   (call t-union (cdr x) y))
                             ))
                 )

```

```

t-member(x y)= (if (eq y nil) then 4
                 else
                 (if (eq x (car y)) then 8
                     else (add 11 (call t-member x (cdr y))))
                 ))

```

```

member(x,y) = (if (eq y nil) then false
               else
               (if (eq x (car y)) then true
                   else (call member x (cdr y))
               ))

```

Time bound program. The goal is now to construct the time bound program. The algorithm to be described in chapter 4 requires that we specify what measure should be used for size of the input. With this program it is natural to use the length of lists for both arguments, and the algorithm as it is described in chapter 4 and 5 will then produce:

```
tb (x,y)      = (call t-union (call length-1 x) (call length-1 y))
```

```
length-1 (x) = (if (eq x 0) then nil
                  else (cons 'all (call length-1 (sub x 1))))
```

```
t-union(x,y) = (if (eq x nil) then 4
                  else (add 14
                           (add (call t-member (car x) y)
                                (call t-union (cdr x) y)
                                )
                           )
                  )
```

```
t-member (x y)= (if (eq y nil) then 4
                   else (add 11 (call t-member x (cdr y))))
```

The program contains a function `length-1` that produces lists with the symbol `all`. Length of lists can in Lisp be found by the function

```
length (x) = (if (null x) then 0
                else (add 1 (call length (cdr x))))
```

The function `length-1` is, in a way we define in chapter 4, the *inversion* of that function. In the resulting time bound program above the `all` symbol is just an atom as all the others, but in an intermediate semantics it denotes the set of all S-expressions. Hence a list of `all` symbols will then denote the set of all S-expressions with this length. The intermediate semantics described in chapter 4 treats the symbol `all` in a special way. As example will the equivalence between two `all` symbols be `all`, denoting that the test can be both true and false. In this semantics the time bound program is just the structural composition of the step counting version and the `length-1` function. It is proven correct by showing that the semantics is a safe approximation to the collecting semantics.

A time bound program can now be obtained by compiling from the special semantics to the standard semantics. This can be done straightforwardly but to make the program efficient we use information gained in a data flow analysis. The analysis gives a description of the range and domain for each function in the program, and it can be used to locate expressions that might evaluate to the value `all`. The data flow analysis is described in chapter 5.

The function `member` was in the context reduced to

```
member(e,s) = (if (eq s nil) then false else 'all)
```

With this the `t-union` function transforms to

```

t-union(x,y) = (if (eq x nil) then 4
                 else
                 (if (eq y nil)
                     then (add 14
                            (add (call t-member (car x) y)
                                (call t-union (cdr x) y)
                            )
                     )
                 else
                 (max (add 11
                        (add (call t-member (car x) y)
                            (call t-union (cdr x) y)
                        )
                      )
                     (add 14
                        (add (call t-member (car x) y)
                            (call t-union (cdr x) y)
                        )
                     )
                 ))
                )

```

and this definition can easily and locally be transformed to the definition above.

Optimizations The program above computes the time bound function, but it is unnecessarily large and it can be considerably simplified by unfolding and solving difference equations. The first step is to remove intermediate lists produced by the `length-1` function:

```

tb (x,y)      = (call t-union x y)

t-union(x,y) = (if (eq x 0) then 4
                 else (add 14 (add (call tb (sub x 1) y)
                                   (call t-member y) )))

t-member(y)   = (if (eq y 0) then 4
                 else (add 11 (call t-member (sub y 1))))

```

These difference equations are easily solved giving:

```

tb (x,y) = (add 4 (add (mul 19 x) (mul 11 (mul x y))))

```

The transformation system that removes the internal list are described in chapter 6, and in chapter 7 we give rules for solving or optimizing finite-difference equations.

The final time bound program actually computes the maximal running time. It is in fact the exact running time of the `union` program with two disjoint sets as arguments.

Chapter 3

Semantics of Step Counting Versions

3.1 Language.

Programs will be expressed in a first order subset of Lisp, containing list and number processing. A program consists of a number of function definitions:

$$F_1(x_1, \dots, x_k) = \langle exp \rangle \& \dots \& F_n(x_1, \dots, x_k) = \langle exp \rangle$$

where $\langle exp \rangle$ is an expression in the language. Expressions are built from conditionals, function calls, basic operations, constants and variables:

$$\begin{aligned} \langle exp \rangle ::= & \text{ (if } \langle exp \rangle \text{ then } \langle exp \rangle \text{ else } \langle exp \rangle) \\ & | \text{ (call } \langle function\ name \rangle \langle exp \rangle^*) \\ & | \text{ (} A_i \langle exp \rangle^*) \\ & | \text{ '}\langle any\ atom \rangle \\ & | \text{ } x_1 \text{ | } \dots \text{ | } x_k \end{aligned}$$

As basic operations (A_i) we accept the kernel of built-in lisp functions : `cons`, `car`, `cdr`, `eq`, `atom`, `add`, `sub`, ...

`eq` tests equality between atoms, and it is false if one argument is not an atom. The set of values is S-expressions from Lisp. A value is either an atom or a dotted pair of S-expressions. The set of atoms should at least contain integers, `nil`, `true`, and `false`. Numbers and these three atoms are accepted as being self-quoting.

The semantics is the obvious one with call-by-value and static scoping, and it is described formally in the next section. For notational convenience in the formal semantics we assume that functions and basic operations have the same number of arguments - let's say k arguments. Since programs and time bound programs are

syntactic objects we must distinguish between programs and the functions they compute. This is done by a semantics function:

$$\mathbf{L} : \text{programs} \rightarrow V_{\perp}^k \rightarrow V_{\perp}$$

where *programs* is the set of programs described above and V_{\perp} is the set of S-expressions extended with an undefined element \perp . Then $\mathbf{L} \mathbf{p} \langle x_1, \dots, x_k \rangle$ is the value obtained from calling the first function in the program \mathbf{p} with input $\langle x_1, \dots, x_k \rangle$. The function \mathbf{L} can be defined formally by a denotational semantics. This is done in the next section, and the function \mathbf{L} is then just:

$$\mathbf{L} \mathbf{p} = \mathbf{U}[\mathbf{p}] \downarrow 1$$

`Typewriter script` will be used in names for programs and in program text and function names are *italicized*.

3.2 Semantics.

The semantics of the language L can be described as an interpretation (see [Mycroft & Nielson, 1983] and [Jones & Mycroft, 1986]) of a general framework. This interpretation is the Standard Interpretation given as a direct semantics.

The semantic domains are

- D : Value denotations
- Φ : Function denotation

The variables used are

- $\phi \in \Phi^n$: Function environment
- $\nu \in D^k$: Variable environment

The semantic functions have functionality

$$\mathbf{E}[\text{expression}] : \Phi^n \rightarrow D^k \rightarrow D$$

$$\mathbf{U}[\text{program}] : \Phi^n$$

The semantic scheme is

$$\begin{aligned} \mathbf{E}[Xi]\nu\phi &= \text{fetch}_i(\nu) \\ \mathbf{E}[a]\nu\phi &= \text{atom}(a) \\ \mathbf{E}[(A_i E_1, \dots, E_k)]\nu\phi &= \text{basic}_i(\phi, \mathbf{E}[E_1]\nu\phi, \dots, \mathbf{E}[E_k]\nu\phi) \\ \mathbf{E}[(\text{call } F_i E_1, \dots, E_k)]\nu\phi &= \text{apply}_i(\phi, \mathbf{E}[E_1]\nu\phi, \dots, \mathbf{E}[E_k]\nu\phi) \\ \mathbf{E}[(\text{if } E_1 \text{ then } E_2 \text{ else } E_3)]\nu\phi &= \text{cond}(\mathbf{E}[E_1]\nu\phi, \mathbf{E}[E_2]\nu\phi, \mathbf{E}[E_3]\nu\phi) \end{aligned}$$

$$\mathbf{U}\llbracket F_1(x_1, \dots, x_k) = E_1 \& \dots \& F_n(x_1, \dots, x_k) = E_n \rrbracket = \\ \text{fix}(\lambda\phi.\text{iterate}(\mathbf{E}\llbracket E_1 \rrbracket\phi, \dots, \mathbf{E}\llbracket E_n \rrbracket\phi))$$

Standard interpretation Let D be the flat domain of S-expressions V_\perp and let $\Phi : D^k \rightarrow D$ be the function environment. The following functions are continuous.

$$\begin{aligned} \text{fetch}_i(\nu) &= \nu_i \\ \text{atom}(a) &= a \\ \text{apply}_i(\phi, e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\ &\phi_i(e_1, \dots, e_k) && \text{otherwise} \\ \text{basic}_i(\phi, e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\ &op_i(e_1, \dots, e_k) && \text{otherwise} \\ \text{cond}(e_1, e_2, e_3) &= e_2 && \text{if } e_1 = \text{true} \\ &e_3 && \text{if } e_1 = \text{false} \\ &\perp && \text{otherwise} \\ \text{iterate}(\phi_1, \dots, \phi_k) &= \langle \phi_1, \dots, \phi_k \rangle \end{aligned}$$

3.3 Step Counting Version

The “running time” of programs with given input is in general a machine dependent property. It is convenient to define it in a machine independent way. This can be done by defining the running time as the number of basic operations to be performed under the evaluation or equivalent as the length of the computation sequence (see [Talcott, 1986]). For most implementations the real computation time can be found by multiplying with a machine dependent constant factor. In [Kasai et al., 1980] the time measures are computed directly in microseconds distinguishing between the running time of different basic operations. Wegbreit uses also different execution times for the basic operations by constructing expressions containing (unspecified) constants for these execution times. [Metayer, 1985] ignores the basic operations and counts only the number of function calls.

The program to count the number of performed basic operations is called a *step counting version* \mathfrak{t}_p of \mathfrak{p} . The name is taken from [Adachi, 1979], but it is a well known concept in the literature ([Wegbreit, 1975] : Local cost assignment, [Metayer, 1985] : Recursive complexity function). The step counting version of a program \mathfrak{p} is a program \mathfrak{t}_p , which called with the same arguments as \mathfrak{p} computes the number of basic operations \mathfrak{p} will perform in evaluating its value (and loops if \mathfrak{p} does). The simple syntax-directed translation scheme to construct a step counting

$$\begin{aligned}
\mathbf{T} &: \text{programs} \rightarrow \text{programs} \\
\mathbf{T}[[X_i]] &= [1] \\
\mathbf{T}[[A_1]] &= [1] \\
\mathbf{T}[[A_i E_1, \dots, E_k]] &= [(\text{add } 1] \mathbf{T}[[E_1]] \dots \mathbf{T}[[E_k]] \text{)}] \\
\mathbf{T}[[\text{call } F_i E_1, \dots, E_k]] &= [(\text{add}(\text{call } T_i E_1, \dots, E_k)] \mathbf{T}[[E_1]] \dots \mathbf{T}[[E_k]] \text{)}] \\
\mathbf{T}[[\text{if } E_1 \text{ then } E_2 \text{ else } E_3]] &= [(\text{if } E_1 \text{ then } (\text{add}] \mathbf{T}[[E_1]] \mathbf{T}[[E_2]] \text{)}] \\
&\quad [\text{else } (\text{add}] \mathbf{T}[[E_1]] \mathbf{T}[[E_3]] \text{)}] \\
\mathbf{T}[[F_1(X_1, \dots, X_k) = E_1 \& \dots \& F_n(X_1, \dots, X_k) = E_n]] \\
&= [T_1(X_1, \dots, X_k) =] \mathbf{T}[[E_1]] \quad \& \dots \& \quad [T_n(X_1, \dots, X_k) =] \mathbf{T}[[E_n]] \& \\
&\quad [F_1(X_1, \dots, X_k) = E_1 \quad \& \dots \& \quad F_n(X_1, \dots, X_k) = E_n]
\end{aligned}$$

Figure 3.1: Construction of a step counting version of programs.

version is given in figure 3.1 where quasiquotes ($[$ and $]$) are used to build the new program text. Note that \mathbf{t}_p contains both the functions of p and a new time function T_i for each function F_i of p . The syntax of the output programs is formally defined in the next section and we then define

Definition 3.1. A program \mathbf{t}_p is a step counting version of a program p if \mathbf{t}_p is the result of using the translation scheme in figure 3.1 on the program p .

This definition is just one of many possibilities. The "ones" in the translation scheme could be replaced by concrete figures from a specific machine implementation, or we could only count the number of *cons* operations performed in computing the result.

Instead of transforming the program to incorporate expressions to update the time used so far, the program could be abstractly interpreted as evaluating to a tuple describing the result plus the time used. In [Nielson, 1985] an idea similar to this is used in a proof system for run-time analysis of programs. It is argued that the modifications makes it difficult to prove that the result of the analysis is a statement about the original language. [Talcott, 1986] gives a general framework to derive correct modified programs from the original programs. The framework can be used to derive programs to show other internal properties; properties of the computational behavior not apparent from the result of the computation.

The use of the step counting version as a basis for the construction of time bound programs allows us to use the same method in the analysis of other internal

$$\begin{aligned}
E_\tau &\rightarrow 'n \\
&| (\text{add } E_{\tau 1}, \dots, E_{\tau k}) \\
&| (\text{call } T_i E_{\sigma 1}, \dots, E_{\sigma k}) \\
&| (\text{if } E_{\sigma 1} \text{ then } E_{\tau 2} \text{ else } E_{\tau 3}) \\
\\
E_\sigma &\rightarrow X_i \\
&| 'a \\
&| (A_i E_{\sigma 1}, \dots, E_{\sigma k}) \\
&| (\text{call } F_i E_{\sigma 1}, \dots, E_{\sigma k}) \\
&| (\text{if } E_{\sigma 1} \text{ then } E_{\sigma 2} \text{ else } E_{\sigma 3}) \\
\\
Pgm &\rightarrow T_1(X_1, \dots, X_k) = E_{\tau 1} \& \dots \& T_n(X_1, \dots, X_k) = E_{\tau n} \\
&F_1(X_1, \dots, X_k) = E_{\sigma 1} \& \dots \& F_n(X_1, \dots, X_k) = E_{\sigma n} \&
\end{aligned}$$

Figure 3.2: Abstract Syntax

properties. It makes also the analysis language independent. If we should make time analysis for a Pascal program we will only need to construct a step counting version of the Pascal program in the Lisp subset language. All what then is needed is to simulate the data structures of Pascal in Lisp. How this is done will not change the time bound function but only the efficiency of the time bound program.

3.4 A Semantic Framework.

Step counting versions of programs will belong to an extended syntax, where two *sorts* (σ and τ) are separated in the syntax. There are two types of expressions : E_σ evaluating to S-expressions and E_τ evaluating to time values. We will use a general framework parallel to the one of section 3.2 to describe the semantics of programs in this extended syntax. This framework will be used with many interpretations in the following program analysis, and it is a simple extension of the framework used in [Jones & Mycroft, 1986]. The description will contain seven

sets, but not all sets are used in every interpretation:

- D_σ : Data values (Carrier) of sort σ
- D_τ : Data values (Carrier) of sort τ – not used
- Φ_σ : Function denotations of sort σ
- Φ_τ : Function denotations of sort τ
- C : Input specification of sort σ
- R_σ : Expression results of sort σ
- R_τ : Expression results of sort τ

There are three semantic variables

- $\nu \in D_\sigma^k$: Variable environment
- $\theta \in \Phi_\tau^k$: Function environment of sort τ
- $\phi \in \Phi_\sigma^n$: Function environment of sort σ

There are now three semantics function

- E** $_\sigma$: $\Phi_\tau^n \rightarrow \Phi_\sigma^n \rightarrow D_\sigma^k \rightarrow R_\sigma$
- E** $_\tau$: $\Phi_\tau^n \rightarrow \Phi_\sigma^n \rightarrow D_\sigma^k \rightarrow R_\tau$
- U** : $\Phi_\tau^n \times \Phi_\sigma^n$

An interpretation consists of defining the six sets and a number of auxiliary functions

- $atom_\tau$: $Atoms \rightarrow R_\tau$
- add_τ : $R_\tau^k \rightarrow R_\tau$
- $apply_{\tau i}$: $\Phi_\tau^n \times R_\sigma^k \rightarrow R_\tau$
- $cond_\tau$: $R_\sigma \times R_\tau \times R_\tau \rightarrow R_\tau$
- $fetch_{\sigma i}$: $D_\sigma^k \rightarrow R_\sigma$
- $atom_\sigma$: $Atoms \rightarrow R_\sigma$
- $basic_{\sigma i}$: $\Phi_\sigma^n \times R_\sigma^k \rightarrow R_\sigma$
- $apply_{\sigma i}$: $\Phi_\sigma^n \times R_\sigma^k \rightarrow R_\sigma$
- $cond_\sigma$: $R_\sigma \times R_\sigma \times R_\sigma \rightarrow R_\sigma$
- $init$: $C^n \times \Phi_\tau^n \times \Phi_\sigma^n \rightarrow \Phi_\tau^n \times \Phi_\sigma^n$
- $iterate$: $\Phi_\tau^n \times \Phi_\sigma^n \times (D_\sigma^k \rightarrow R_\tau)^n \times (D_\sigma^k \rightarrow R_\sigma)^n \rightarrow \Phi_\tau^n \times \Phi_\sigma^n$

These functions are used in the following scheme:

$$\begin{aligned}
\mathbf{E}_\tau \llbracket 'n \rrbracket \theta \phi \nu &= \text{atom}_\tau(n) \\
\mathbf{E}_\tau \llbracket (\text{add } E_1, \dots, E_k) \rrbracket \theta \phi \nu &= \text{add}(\mathbf{E}_\tau \llbracket E_1 \rrbracket \theta \phi \nu, \dots, \mathbf{E}_\tau \llbracket E_k \rrbracket \theta \phi \nu) \\
\mathbf{E}_\tau \llbracket (\text{call } T_i E_1, \dots, E_k) \rrbracket \theta \phi \nu &= \text{apply}_{\tau i}(\theta, \mathbf{E}_\sigma \llbracket E_1 \rrbracket \theta \phi \nu, \dots, \mathbf{E}_\sigma \llbracket E_k \rrbracket \theta \phi \nu) \\
\mathbf{E}_\tau \llbracket (\text{if } E_1 \text{ then } E_2 \text{ else } E_3) \rrbracket \theta \phi \nu &= \text{cond}_\tau(\mathbf{E}_\sigma \llbracket E_1 \rrbracket \theta \phi \nu, \mathbf{E}_\tau \llbracket E_2 \rrbracket \theta \phi \nu, \mathbf{E}_\tau \llbracket E_3 \rrbracket \theta \phi \nu) \\
\mathbf{E}_\sigma \llbracket X_i \rrbracket \theta \phi \nu &= \text{fetch}_i(\nu) \\
\mathbf{E}_\sigma \llbracket 'a \rrbracket \theta \phi \nu &= \text{atom}_\sigma(a) \\
\mathbf{E}_\sigma \llbracket (A_i E_1, \dots, E_k) \rrbracket \theta \phi \nu &= \text{basic}_i(\phi, \mathbf{E}_\sigma \llbracket E_1 \rrbracket \theta \phi \nu, \dots, \mathbf{E}_\sigma \llbracket E_k \rrbracket \theta \phi \nu) \\
\mathbf{E}_\sigma \llbracket (\text{call } F_i E_1, \dots, E_k) \rrbracket \theta \phi \nu &= \text{apply}_{\sigma i}(\phi, \mathbf{E}_\sigma \llbracket E_1 \rrbracket \theta \phi \nu, \dots, \mathbf{E}_\sigma \llbracket E_k \rrbracket \theta \phi \nu) \\
\mathbf{E}_\sigma \llbracket (\text{if } E_1 \text{ then } E_2 \text{ else } E_3) \rrbracket \theta \phi \nu &= \text{cond}_\tau(\mathbf{E}_\sigma \llbracket E_1 \rrbracket \theta \phi \nu, \mathbf{E}_\sigma \llbracket E_2 \rrbracket \theta \phi \nu, \mathbf{E}_\sigma \llbracket E_3 \rrbracket \theta \phi \nu)
\end{aligned}$$

$$\begin{aligned}
\mathbf{U} \llbracket T_1(X_1, \dots, X_k) = E_{\tau 1} \&\dots\& T_n(X_1, \dots, X_k) = E_{\tau n} \& \\
F_1(X_1, \dots, X_k) = E_{\sigma 1} \&\dots\& F_n(X_1, \dots, X_k) = E_{\sigma n} \rrbracket \mathbf{c} \\
&= \text{fix } \lambda \theta \phi. \text{iterate}(\text{init}(\mathbf{c}, \theta, \phi), \mathbf{E}_\sigma \llbracket E_{\sigma 1} \rrbracket \text{init}(\mathbf{c}, \theta, \phi), \dots, \mathbf{E}_\sigma \llbracket E_{\sigma n} \rrbracket \text{init}(\mathbf{c}, \theta, \phi), \\
&\quad \mathbf{E}_\tau \llbracket E_{\tau 1} \rrbracket \text{init}(\mathbf{c}, \theta, \phi), \dots, \mathbf{E}_\tau \llbracket E_{\tau n} \rrbracket \text{init}(\mathbf{c}, \theta, \phi))
\end{aligned}$$

Standard Interpretation.

The Standard Interpretation of the language is denoted \mathbf{U}_S and the sets are :

$$\begin{aligned}
D_\sigma &= V_\perp && \text{The (flat) domain of S-expression} \\
D_\tau &= N_\perp && \text{The (flat) domain of natural numbers} \\
\Phi_\sigma &= V_\perp^k \rightarrow V_\perp \\
\Phi_\tau &= V_\perp^k \rightarrow N_\perp \\
C &&& \text{--- not used} \\
R_\sigma &= D_\sigma \\
R_\tau &= D_\tau
\end{aligned}$$

The auxiliary functions are

$$\begin{aligned}
atom_\tau(n) &= n && \text{if } n \in N \\
&\perp && \text{otherwise} \\
add_\tau(e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\
&e_1 + \dots + e_k && \text{otherwise} \\
cond_\tau(e_1, e_2, e_3) &= e_2 && \text{if } e_1 = true \\
&e_3 && \text{if } e_1 = false \\
&\perp && \text{otherwise} \\
apply_{\tau_i}(\theta, e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\
&\theta_i(e_1, \dots, e_k) && \text{otherwise} \\
fetch_{\sigma_i}(\nu) &= \nu_i \\
atom_\sigma(a) &= 'a \\
apply_{\sigma_i}(\phi, e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\
&\phi_i(e_1, \dots, e_k) && \text{otherwise} \\
basic_{\sigma_i}(\phi, e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\
&op_i(e_1, \dots, e_k) && \text{otherwise} \\
cond_\sigma(e_1, e_2, e_3) &= e_2 && \text{if } e_1 = true \\
&e_3 && \text{if } e_1 = false \\
&\perp && \text{otherwise} \\
init(\mathbf{c}, \theta, \phi) &= (\theta, \phi) \\
iterate(\theta, \phi, t_1, \dots, t_k, f_1, \dots, f_k) &\langle t_1, \dots, t_k, f_1, \dots, f_k \rangle
\end{aligned}$$

Definition 3.2. The semantics of a step counting version \mathfrak{t}_p of a program p is defined by the semantics function \mathbf{L}_S :

$$\mathbf{L}_S \mathfrak{t}_p = \mathbf{U}_S[\mathfrak{t}_p] \downarrow 1$$

where $\mathfrak{t}_p = \mathbf{T}[p]$.

Chapter 4

Time Bound Programs

In this chapter we describe and prove the correctness of the algorithm to construct time bound programs.

Definition 4.1. Given a program p and an input-size measure $length$. tb_p is then a time bound program if and only if it satisfies

$$\mathbf{L} \, tb_p \langle n_1, \dots, n_k \rangle \geq \max \{ \mathbf{L} \, t_p \langle x_1, \dots, x_k \rangle \mid \forall x_i : length(x_i) = n_i \}$$

where t_p is the step counting version of p and “ \geq ” is a partial correctness ordering : if the left hand side is defined then the right hand side is defined and the left hand side will be not less than the right hand side.

In other words the time bound program should give a partially correct upper bound to the running time.

Construction 4.2. In this chapter we construct a program tbl , which in a special semantics $\tilde{\mathbf{L}}$ gives an upper bound of the running time. The program tbl is just the structural composition of t_p and a function $length$ which in a sense we define in section 4.6 computes the inversion of the size function $length : \mathbf{length-1}$

Outline of Proof.

The semantics $\tilde{\mathbf{L}}$ is based on an extension of the set of S-expressions. The new set \tilde{V}_\perp is S-expressions extended with a new symbol all , that can appear everywhere in a structure. The denotation of an S-expression with all is a set including every value obtainable by replacing every all by an arbitrary S-expression. As example will ' all ' represent the set V of all S-expressions and ' $(all \, all)$ ' represents all S-expressions of length 2. The set \tilde{V}_\perp is defined formally in section 4.2 and the semantics $\tilde{\mathbf{L}}$ is described in section 4.3 and 4.4.

The elements of \tilde{V}_\perp describe sets of S-expressions, and we show that it defines an image of the powerset $\mathbf{P}(V)$ by two functions

$$\begin{aligned}\alpha : \mathbf{P}(V) &\rightarrow \tilde{V}_\perp \\ \gamma : \tilde{V}_\perp &\rightarrow \mathbf{P}(V)\end{aligned}$$

and that the semantics $\tilde{\mathbf{L}}$ is a safe approximation of the collecting semantics \mathbf{L}_C .

The collecting semantics is described in section 4.1 and in section 4.4 is $\tilde{\mathbf{L}}$ shown to be a safe approximation. The collecting semantics is also shown to be a proper extension of the standard semantics \mathbf{L}_S .

The proof is structured in the following way:

1. The definition of a time bound program can be restated in the collecting semantics:

$$\mathbf{L}_S \mathbf{tb}_p \langle n_1, \dots, n_k \rangle \geq \max \mathbf{L}_C \mathbf{t}_p \langle \mathit{length}^{-1}(n_1), \dots, \mathit{length}^{-1}(n_k) \rangle$$

where $\mathit{length}^{-1} : N \rightarrow \mathbf{P}(V)$ is the inversion of the size measure.

2. We construct a function $\mathbf{length-1}$ that satisfies

$$\gamma(\tilde{\mathbf{L}} \mathbf{length-1} \langle n \rangle) \supseteq \mathit{length}^{-1}(n)$$

3. With $\mathbf{tb1}$ as the structural composition of \mathbf{t}_p and $\mathbf{length-1}$ we get that

$$\tilde{\mathbf{L}} \mathbf{tb1} \langle n_1, \dots, n_k \rangle = \tilde{\mathbf{L}} \mathbf{t}_p \langle \tilde{\mathbf{L}} \mathbf{length-1} \langle n_1 \rangle, \dots, \tilde{\mathbf{L}} \mathbf{length-1} \langle n_k \rangle \rangle$$

4. The $\tilde{\mathbf{L}}$ semantics is a safe approximation of the collecting semantics.

$$\tilde{\mathbf{L}} \mathbf{t}_p \langle x_1, \dots, x_k \rangle \geq \max \mathbf{L}_C \mathbf{t}_p \langle \gamma(x_1), \dots, \gamma(x_k) \rangle$$

Hence the program $\mathbf{tb1}$ can be proven to be a safe time bound program.

5. Finally it is possible to compile the program $\mathbf{tb1}$ to the standard semantics, but to make effective target programs the compilation uses information from a data flow analysis presented in the next chapter.

For $k = 1$ we can combine the proof in the following diagram:

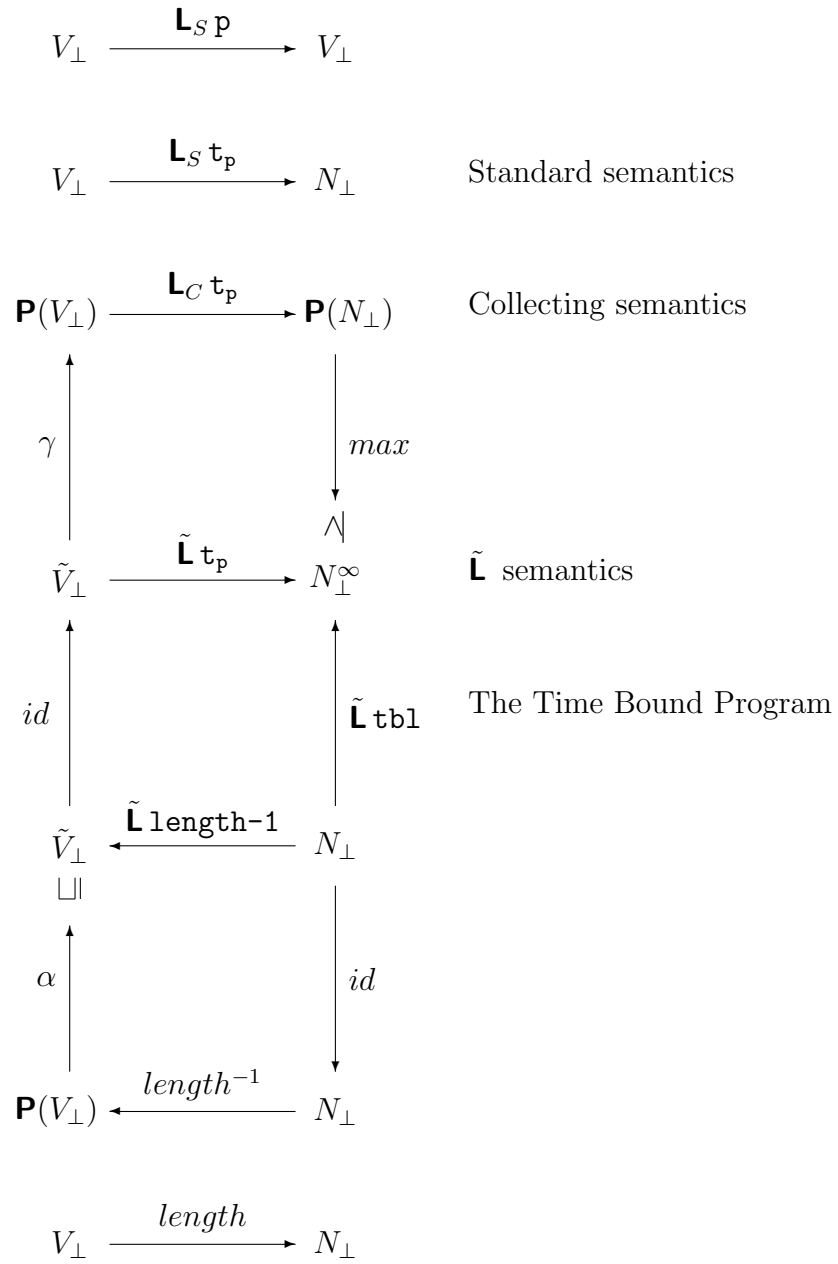


Figure 4.1: Construction of time bound function.

4.1 Collecting Semantics

The collecting semantics (called the static semantics in [Cousot & Cousot, 1977]) interprets the program on sets of data. A Collecting semantics \mathbf{U}_C can be defined by

$$\mathbf{U}_C[[pgm]] \downarrow i = \lambda S_1, \dots, S_k. \{ \mathbf{U}_S[[pgm]] \downarrow i (x_1, \dots, x_k) \mid x_j \in S_j \} \quad i = 1, \dots, 2n$$

but to use it in proofs we will specify it as an interpretation of the semantics framework described in chapter 3.

The semantics will be of the so-called *independent attribute method* (see [Nielson, 1984]) meaning that we only give an upper bound of the range of a program function given an input set. This is because different variables are treated independently. More technically we are modeling the powerset $\mathbf{P}(V)^k$ instead of $\mathbf{P}(V^k)$. The reason for the use of this type of collecting semantics is that it is technically simpler and the extra information is not needed. The collecting semantics is only used in the proofs to relate the standard semantics (section 3.4) to the \tilde{V}_\perp -semantics (section 4.4). The \tilde{V}_\perp -semantics is shown to be a safe approximation to the collecting semantics. The collecting semantics is itself seen to be a safe approximation to the most precise so-called *relational collecting semantics* (see [Nielson, 1984]).

As a collecting semantics works on sets of data we need powersets in the definition. Collecting semantics will also be used in the next chapter, but with powersets of more complicated domains than here. The definition of powersets will therefore be more precise than the flat domains calls for:

Hoare Powerdomains.

The use of powerdomains in semantics originates from non-deterministic languages, but in recent years they have also been used in abstract interpretation. In [Mycroft & Nielson, 1983] a variant of the Plotkin powerdomain is used for strong abstract interpretation. [Nielson, 1984] uses Hoare powerdomain (there called relational powerdomain) in a collecting semantics for a metalanguage for denotational semantics. In [Burn, Hankin & Abrahamsky, 1986] Hoare powerdomain is used in strictness analysis. The Hoare powerdomain has been seen to be the best for this application. More complete treatments of Hoare powerdomains can be found in [Sestoft & Søndergaard, 1986], [Burn, Hankin & Abrahamsky, 1986], and [Nielson, 1984]

If D is a domain, the Hoare powerdomain is defined as the set $\wp(D)$ where

$$\begin{aligned} X \in \wp(D) \Leftrightarrow \quad & Y \sqsubseteq X, Y \text{ directed} \Rightarrow \bigsqcup Y \in X \\ & \wedge y \sqsubseteq x \in X \Rightarrow y \in X \end{aligned}$$

Sets that satisfy this condition are said to be Scott-closed, and the least Scott-closed set containing a set X is denoted $\mathbf{SC}(X)$. The lower (or left) closure of a

set is

$$\mathbf{LC}(X) = \{y \in D \mid \exists x \in X : y \sqsubseteq x\}$$

and this is just the second part of the condition for a set to be Scott-closed. The set $\wp(D)$ is equipped with the set inclusion ordering. The main result is that $(\wp(D), \subseteq)$ is a domain with least element $\{\perp\}$ where \perp is the least element in D . In the following we use the notation $\wp(D)$ for the powerdomain constructor, and $\mathbf{P}(D)$ for the powerset constructor. If D is a flat domain, then will $\wp(D)$ be the set of subsets of D , that contains the least element \perp :

$$\wp(D) = \{X \in \mathbf{P}(D) \mid \perp \in X\}$$

This means that the powerdomain $\wp(V_\perp)$ of a flat domain is isomorphic with the powerset $\mathbf{P}(V)$.

A function $f : D \rightarrow E$ where D and E are domains, can be extended to a function $f_* : \wp(D) \rightarrow \wp(E)$ by

$$f_*(X) = \mathbf{SC}(\{f(x) \mid x \in X\})$$

This function will be continuous and it will be called the Hoare extension of the function f .

For a flat domain V_\perp a function $f : V_\perp \rightarrow V_\perp$ is extended to a function $f_* : \wp(V_\perp) \rightarrow \wp(V_\perp)$ by

$$\begin{aligned} f_*(X) &= \mathbf{SC}(\{f(x) \mid x \in X\}) \\ &= \{f(x) \mid x \in X\} \cup \{\perp\} \\ &= f_{1*}(X) \cup \{\perp\} \end{aligned}$$

where

$$\begin{aligned} f_{1*} &: PV \rightarrow PV \\ f_{1*}(X) &= \{f(x) \mid x \in X \cup \{\perp\} \wedge f(x) \neq \perp\} \end{aligned}$$

The last function is an extension of function to the powerset stripped from the bottom element. This way to extend function can for example be found in [Reynolds, 1969] and it is isomorphic with the Hoare extension.

For a domain D we define a map

$$\{\ast\} : D \rightarrow \wp(D)$$

by

$$\{d\} = \mathbf{LC}(\{d\})$$

Notice that the lower closure of a singleton set also will be Scott-closed. The function satisfies the two conditions

1. $\{\} * \}$ is continuous.
2. For $f : D \rightarrow E$, $f_* \circ \{\} * \}_D = \{\} * \}_E \circ f$

From a technical point of view it is for a flat domain V_\perp , N_\perp a bit simpler to work with the stripped powerset $\mathbf{P}(V)$, $\mathbf{P}(N)$ in stead of the Hoare powerdomain $\wp(V_\perp)$, $\wp(N_\perp)$. The isomorphism secures the domain properties, so it is only a matter of taste what to choose. The collecting semantics described here will be based on power sets, but the definition by Hoare powerdomains would be very similar.

Collecting Interpretation.

The Collecting Interpretation of the language is denoted \mathbf{U}_C and the sets are :

D_σ	$= \mathbf{P}(V)$	The powerset of S-expression
D_τ	$= \mathbf{P}(N)$	The powerset of natural numbers
Φ_σ	$= \mathbf{P}(V)^k \rightarrow \mathbf{P}(V)$	
Φ_τ	$= \mathbf{P}(V)^k \rightarrow \mathbf{P}(N)$	
C		– not used
R_σ	$= D_\sigma$	
R_τ	$= D_\tau$	

The auxiliary functions are

$$\begin{aligned}
atom_\tau(n) &= \{n\} && \text{if } n \in N \\
& && \emptyset && \text{otherwise} \\
add_\tau(e_1, \dots, e_k) &= \emptyset && \text{if some } e_i = \emptyset \\
& && \{x_1 + \dots + x_k \mid x_i \in e_i\} && \text{otherwise} \\
cond_\tau(e_1, e_2, e_3) &= e_2 \cup e_3 && \text{if } true \in e_1 \wedge false \in e_1 \\
& && e_2 && \text{if } true \in e_1 \\
& && e_3 && \text{if } false \in e_1 \\
& && \emptyset && \text{otherwise} \\
apply_{\tau_i}(\theta, e_1, \dots, e_k) &= \emptyset && \text{if some } e_i = \emptyset \\
& && \theta_i(e_1, \dots, e_k) && \text{otherwise} \\
fetch_{\sigma_i}(\nu) &= \nu_i \\
atom_\sigma(a) &= \{a\} \\
apply_{\sigma_i}(\phi, e_1, \dots, e_k) &= \emptyset && \text{if some } e_i = \emptyset \\
& && \phi_i(e_1, \dots, e_k) && \text{otherwise} \\
basic_{\sigma_i}(\phi, e_1, \dots, e_k) &= \emptyset && \text{if some } e_i = \emptyset \\
& && \{op(x_1, \dots, x_k) \mid x_i \in e_i \wedge op(x_1, \dots, x_k) \neq \perp\} && \text{otherwise} \\
cond_\sigma(e_1, e_2, e_3) &= e_2 \cup e_3 && \text{if } true \in e_1 \wedge false \in e_1 \\
& && e_2 && \text{if } true \in e_1 \\
& && e_3 && \text{if } false \in e_1 \\
& && \emptyset && \text{otherwise} \\
init(\mathbf{c}, \theta, \phi) &= (\theta, \phi) \\
iterate(\theta, \phi, t_1, \dots, t_k, f_1, \dots, f_k) &= \langle t_1, \dots, t_k, f_1, \dots, f_k \rangle
\end{aligned}$$

Correctness (sketch).

The collecting semantics was based on powersets instead of Hoare powerdomain. This means, that we will redefine the injection function $\{\!*\!\}$ to :

$$\begin{aligned} \{\!*\!\} : V_{\perp} &\rightarrow \mathbf{P}(V) \\ \{\!d\!\} &= \emptyset && \text{if } d = \perp \\ &= \{d\} && \text{otherwise} \end{aligned}$$

We want to show that the collecting semantics \mathbf{U}_C is an extension of the standard semantics \mathbf{U}_S , by proving :

$$\begin{aligned} \forall i = 1, \dots, 2n, \forall x_1, \dots, x_k \in V_{\perp} \\ \mathbf{U}_C \llbracket pgm \rrbracket \downarrow i (\{\!x_1\!\}, \dots, \{\!x_k\!\}) = \{\!\mathbf{U}_S \llbracket pgm \rrbracket \downarrow i (x_1, \dots, x_k)\!\} \end{aligned}$$

We just need to show that for

$$\begin{aligned} \phi &\in (V_{\perp}^k \rightarrow V_{\perp})^n \\ \Phi &\in (\mathbf{P}(V)^k \rightarrow \mathbf{P}(V))^n \\ \theta &\in (V_{\perp}^k \rightarrow N_{\perp})^n \\ \Theta &\in (\mathbf{P}(V)^k \rightarrow \mathbf{P}(N))^n \end{aligned}$$

that

$$\begin{aligned} \{\!*\!\} \circ \phi_i &= \Phi_i \circ \{\!*\!\}^n \wedge \\ \{\!*\!\} \circ \theta_i &= \Theta_i \circ \{\!*\!\}^n \end{aligned}$$

implies

$$\{\!\mathbf{E}_{\sigma S} \llbracket \langle exp \rangle \rrbracket \theta \phi(x_1, \dots, x_k)\!\} = \mathbf{E}_{\sigma C} \llbracket \langle exp \rangle \rrbracket \Phi \Theta(\{\!x_1\!\}, \dots, \{\!x_k\!\})$$

and

$$\{\!\mathbf{E}_{\tau S} \llbracket \langle exp \rangle \rrbracket \theta \phi(x_1, \dots, x_k)\!\} = \mathbf{E}_{\tau C} \llbracket \langle exp \rangle \rrbracket \Phi \Theta(\{\!x_1\!\}, \dots, \{\!x_k\!\})$$

where

$$\{\!*\!\}^n = \lambda x_1, \dots, x_n. \langle \{\!x_1\!\}, \dots, \{\!x_n\!\} \rangle$$

This is fairly easy to see by structural induction on the syntax of expression. The next theorem follows now by fixpoint induction [Bird, 1976] due to the continuity of \mathbf{E}_S and \mathbf{E}_C .

Safeness Theorem 4.3. With

$$\mathbf{L}_C \langle pgm \rangle = \mathbf{U}_C \llbracket \langle pgm \rangle \rrbracket \downarrow 1$$

the collecting interpretation satisfies

$$\{\!\mathbf{L}_S \langle pgm \rangle (x_1, \dots, x_k)\!\} = \mathbf{L}_C \langle pgm \rangle (\{\!x_1\!\}, \dots, \{\!x_k\!\}) \quad \forall x_i \in V_{\perp}$$

and the $\mathbf{L}_C \langle pgm \rangle$ function is continuous.

Corollary 4.4.

$$\begin{aligned} \mathbf{L}_C \langle pgm \rangle (X_1, \dots, X_k) \supseteq \{ \mathbf{L}_S \langle pgm \rangle (x_1, \dots, x_k) \mid x_i \in X_i \} \\ \forall X_i \in \mathbf{P}(V) \end{aligned}$$

The collecting semantics is simpler than the one from [Nielson, 1984] as we only need subset inclusion and not identity in the corollary. This is due to the use of cartesian products $\mathbf{P}(V) \times \mathbf{P}(V)$ instead of tensor products $\wp(V_\perp) \otimes \wp(V_\perp)$ isomorphic to $\wp(V_\perp \times V_\perp)$.

4.2 The Domain of partly known structures.

The set \tilde{V}_\perp is the set of S-expressions extended with a fresh atom : *all*. If V_\perp satisfies

$$V_\perp = \text{Atoms}_\perp \oplus (V_\perp \otimes V_\perp)$$

then \tilde{V}_\perp satisfies

$$\tilde{V}_\perp = (\text{Atoms} + \{\text{all}\})_\perp \oplus (\tilde{V}_\perp \otimes \tilde{V}_\perp)$$

where \oplus is the coalesced sum and \otimes is the smashed product. This just means, that we don't allow undefined subtrees, but the symbol *all* may appear in a subtree.

Two functions α and γ formally define \tilde{V}_\perp as a model of $\mathbf{P}(V)$.

$$\tilde{V}_\perp \begin{array}{c} \xrightarrow{\gamma} \\ \xleftarrow{\alpha} \end{array} \supseteq \mathbf{P}(V)$$

The function α is called an *abstraction* and the function γ is called a *concretization* (see [Cousot & Cousot, 1977]).

$$\begin{aligned} \gamma(x) = & V && \text{if } x = \text{all}, \text{ else} \\ & \emptyset && \text{if } x = \perp, \text{ else} \\ & \{x\} && \text{if } \text{atom}(x), \text{ else} \\ & \{(a.b) \mid a \in \gamma(x_1) \wedge b \in \gamma(x_2)\} && \text{if } x = (x_1.x_2) \end{aligned}$$

$$\begin{aligned} \alpha(x) = & \perp && \text{if } x = \emptyset, \text{ else} \\ & e && \text{if } x = \{e\}, \text{ else} \\ & \text{all} && \text{if } \text{atomsin}(x), \text{ else} \\ & \text{cons}(\alpha(\text{car}_*(x)), \alpha(\text{cdr}_*(x))) \end{aligned}$$

where '*atomsin*(S)' tests whether the set S contains 1 or more atoms, and '*car*'_*' and '*cdr*'_* are '*car*' and '*cdr*' extended to work elementwise on subsets of V.

Lemma 4.5. The functions α and γ satisfy

- (1) $\alpha(\gamma(x)) = x$, $\forall x \in \tilde{V}_\perp$
- (2) $\gamma(\alpha(S)) \supseteq S$, $\forall S \in \mathbf{P}(V)$

Proof. Symbolic composition of α and γ gives the function

$$\begin{aligned} \alpha\gamma(x) = & \text{all} \quad \text{if } x = \text{all} \quad , \text{ else} \\ & \perp \quad \text{if } x = \perp \quad , \text{ else} \\ & x \quad \text{if } \text{atom}(x) \quad , \text{ else} \\ & \text{cons}(\alpha(\{a \in \gamma(x_1)\}), \alpha(\{b \in \gamma(x_2)\})) \\ & \quad \text{if } x = (x_1.x_2) \end{aligned}$$

or just

$$= x$$

Symbolic composition of γ and α gives

$$\begin{aligned} \gamma\alpha(x) = & \emptyset \quad \text{if } x = \emptyset \quad , \text{ else} \\ & \{e\} \quad \text{if } x = \{e\} \quad , \text{ else} \\ & V \quad \text{if } \text{atomsin}(x) \quad , \text{ else} \\ & \{(a.b) \mid a \in \gamma\alpha(\text{car}_*(S)) \wedge b \in \gamma\alpha(\text{cdr}_*(S))\} \end{aligned}$$

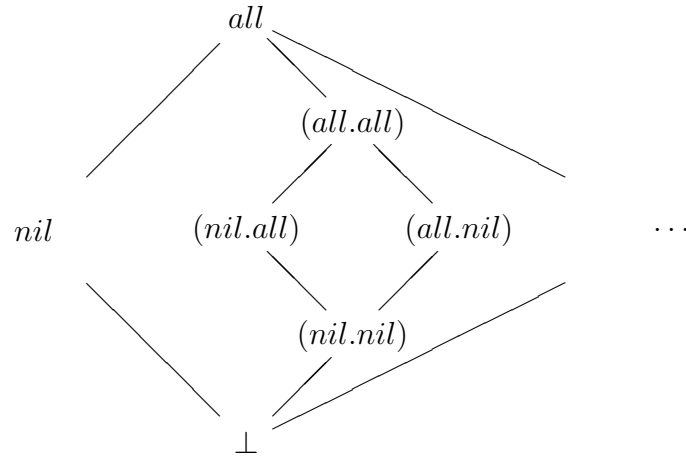
and structural induction shows that

$$\gamma\alpha(S) \supseteq S, \forall S \in \mathbf{P}(V) \square$$

Definition 4.6. The set \tilde{V}_\perp is equipped with an ordering \sqsubseteq induced by the set inclusion ordering of $\mathbf{P}(V)$.

$$x \sqsubseteq y \stackrel{\text{def}}{\iff} \gamma(x) \subseteq \gamma(y) \quad , x, y \in \tilde{V}_\perp$$

Lemma 4.7. The functions α and γ are monotonic.

Figure 4.2: The structure of \tilde{V}_\perp .

Proof. By definition this is true for γ and the function $\gamma\alpha$ (the symbolic composition of γ and α from lemma 4.6) is monotonic, hence

$$\begin{aligned} S_1 \subseteq S_2 &\Leftrightarrow \gamma\alpha(S_1) \subseteq \gamma\alpha(S_2) \\ &\Leftrightarrow \alpha(S_1) \sqsubseteq \alpha(S_2) \quad \square \end{aligned}$$

The functions α and γ are also continuous, but we don't need this property in the proofs.

Definition 4.8. In the set \tilde{V}_\perp we define

$$lub(D) \stackrel{\text{def}}{=} \alpha \left(\bigcup_{y \in D} \gamma(y) \right) \quad D \subseteq \tilde{V}_\perp$$

Lemma 4.8. 4.9. $lub(D)$ exists for all subsets $D \subseteq \tilde{V}_\perp$ and it is the least upper bound (abbreviated to lub).

Proof. We must prove that for any $D \subseteq \tilde{V}_\perp$:

- (1) $\forall x \in D : x \sqsubseteq lub(D)$
- (2) $\forall z \in \tilde{V}_\perp : (\forall x \in D : x \sqsubseteq z) \Leftrightarrow lub(D) \sqsubseteq z$

1. For any $x \in D$ we have :

$$\begin{aligned} \gamma(x) &\subseteq \bigcup_{y \in D} \gamma(y) && \Rightarrow \\ \gamma\alpha(\gamma(x)) &\subseteq \gamma\alpha\left(\bigcup_{y \in D} \gamma(y)\right) && \Rightarrow \\ \gamma(x) &\subseteq \gamma\alpha\left(\bigcup_{y \in D} \gamma(y)\right) && \Rightarrow \\ x &\sqsubseteq \text{lub}(D) \end{aligned}$$

2. For any $z \in \tilde{V}_\perp$ with $\forall x \in D : x \sqsubseteq z$ we have

$$\begin{aligned} \bigcup_{y \in D} \gamma(y) &\subseteq \gamma(z) && \Rightarrow \\ \gamma\alpha\left(\bigcup_{y \in D} \gamma(y)\right) &\subseteq \gamma\alpha(\gamma(z)) && \Rightarrow \\ \text{lub}(D) &\sqsubseteq \alpha(\gamma(z)) = z \end{aligned}$$

Lemma 4.8 shows that the poset $(\tilde{V}_\perp, \sqsubseteq)$ (partially ordered set) is a complete lattice.

The symbol \sqcup is used for binary lubs

$$x \sqcup y = \text{lub}(\{x, y\}) = \alpha(\gamma(x) \cup \gamma(y))$$

and the symbol \bigsqcup is used for least upper bounds of subsets of \tilde{V}_\perp . Where the meaning is clear from the context, the function name 'lub' will be used in both senses. The function is monotonic in both situation due to the monotonicity of α, γ , and union (\cup).

Lemma 4.10. Let $i : V_\perp \rightarrow \tilde{V}_\perp$ be the function $x \rightarrow \alpha(\{x\} \setminus \{\perp\})$ (or $i = \alpha \circ \{\ast\}$) Then

$$\gamma(i(x)) = \{x\} \setminus \{\perp\}, \forall x \in V_\perp$$

Proof. The function $\gamma\alpha$ is the identity mapping for singleton sets. This shows, that \tilde{V}_\perp is both a model of $\mathbf{P}(V)$ and contains all of V . \square

Lemma 4.11. The complete lattice \tilde{V}_\perp has finite height. This means that any monotonic mappings $f : \tilde{V}_\perp \rightarrow X$ are continuous, where X is any domain.

Proof. We will prove that any (ordered) chain in \tilde{V}_\perp must be constant from a certain element. A chain is a sequence of elements of \tilde{V}_\perp : x_1, x_2, \dots where $x_n \sqsubseteq x_{n+1}$ for all $n \geq 1$.

A mapping $n : \tilde{V}_\perp \rightarrow N^\infty$ can be defined by

$$\begin{aligned} n(x) = & \infty && \text{if } x = \perp \\ & 1 && \text{if } x = all \\ & 2 && \text{if } atom(x) \\ & n(x_1) + n(x_2) && \text{if } x = (x_1.x_2) \end{aligned}$$

By structural induction it can be shown that for all $x, y \in \tilde{V}_\perp$:

$$x \sqsubseteq y \Rightarrow x = y \vee n(x) > n(y)$$

Hence, the sequence $n(x_1), n(x_2), \dots$ will be a decreasing chain of positive integers and therefore constant from a certain element. \square

4.3 Extending Functions

The original programs in our language denote functions: $V_\perp^k \rightarrow V_\perp$. In this section the semantics is extended to denote functions: $\tilde{V}_\perp^k \rightarrow \tilde{V}_\perp$. A special treatment for programs yielding integers is necessary even though $N \subseteq V$. With V_\perp and N_\perp as flat domains we want to extend functions $V_\perp^k \rightarrow N_\perp$ to functions $\tilde{V}_\perp \rightarrow N_\perp^\infty$ where N_\perp^∞ is the set of natural numbers and the integer ordering extended with a top element (∞) and a bottom element (\perp) .

Definition 4.12. A function $f : V_\perp^k \rightarrow V_\perp$ can be extended to a function $\tilde{f} : \tilde{V}_\perp^k \rightarrow \tilde{V}_\perp$

$$\tilde{f}(x_1, \dots, x_k) = \alpha(f_*(\gamma(x_1), \dots, \gamma(x_k)))$$

where

$$\begin{aligned} f_*(S_1, \dots, S_k) = \{ & f(x_1, \dots, x_k) \mid x_i \in S_i \cup \{\perp\}, i = 1, \dots, k \\ & \wedge f(x_1, \dots, x_k) \neq \perp \} \end{aligned}$$

$$\begin{array}{ccc}
\tilde{V}_\perp^k & \xrightarrow{\tilde{f}} & \tilde{V}_\perp \\
\downarrow \gamma^k & & \uparrow \alpha \\
\mathbf{P}(V_\perp)^k & \xrightarrow{f_*} & \mathbf{P}(V_\perp) \\
\uparrow \{ * \}^k & & \uparrow \{ * \} \\
V_\perp^k & \xrightarrow{f} & V_\perp
\end{array}$$

Lemma 4.13. For any function $f : V_\perp^k \rightarrow V_\perp$ the extension \tilde{f} will be continuous.

Proof. The function f_* is continuous, as it is isomorphic with the Hoare extension, and so is the composition with continuous functions. \square

That \tilde{f} is a true extension of f is expressed in the next lemma.

Lemma 4.14.

$$\tilde{f}(i(x_1), \dots, i(x_k)) = i(f(x_1, \dots, x_k))$$

where i is defined in lemma 4.9.

It is rather easy to see how the function should be computed.

Basic operations 4.15. Operations $op : V_{\perp}^k \rightarrow V_{\perp}$ can be extended to operations $\tilde{op} : \tilde{V}_{\perp}^k \rightarrow \tilde{V}_{\perp}$. Eg.

$$\begin{aligned}
car(x) &= \perp && \mathbf{if} \ x = \perp \\
& \ x_1 && \mathbf{if} \ x = (x_1.x_2) \\
\tilde{car}(x) &= \perp && \mathbf{if} \ x = \perp \\
& \ all && \mathbf{if} \ x = all \\
& \ x_1 && \mathbf{if} \ x = (x_1.x_2) \\
eq(x, y) &= \perp && \mathbf{if} \ x = \perp \vee y = \perp \vee \\
& && \neg atom(x) \vee \neg atom(y) \\
& \ true && \mathbf{if} \ x = y \\
& \ false && \mathbf{if} \ x \neq y \\
\tilde{eq}(x, y) &= \perp && \mathbf{if} \ x = \perp \vee y = \perp \vee \\
& && \neg atom(x) \vee \neg atom(y) \\
& \ all && \mathbf{if} \ x = all \vee y = all \\
& \ true && \mathbf{if} \ x = y \\
& \ false && \mathbf{if} \ x \neq y \\
cons(x, y) &= \perp && \mathbf{if} \ x = \perp \vee y = \perp \\
& \ (x.y) && \mathbf{otherwise} \\
\tilde{cons}(x, y) &= \perp && \mathbf{if} \ x = \perp \vee y = \perp \\
& \ (x.y) && \mathbf{otherwise}
\end{aligned}$$

The *cons*- operator should combine two lists regardless of *all* values as arguments, whereas the other operations are restrict in *all* (give *all* as result if an argument is *all*).

Conditionals 4.16. The conditionals in the original semantics are defined by the function

$$\begin{aligned}
cond(e_1, e_2, e_3) &= \mathbf{case} \ e_1 \ \mathbf{of} \\
& \ true && : e_2 \\
& \ false && : e_3 \\
& \ \mathbf{otherwise} && : \perp
\end{aligned}$$

The continuous extension is

$$\begin{aligned} \tilde{c\text{ond}}(e_1, e_2, e_3) = & \mathbf{case } e_1 \mathbf{ of} \\ & \mathit{true} \quad : e_2 \\ & \mathit{false} \quad : e_3 \\ & \mathit{all} \quad : \alpha(\gamma(e_2) \cup \gamma(e_3)) \\ & \mathbf{otherwise} \quad : \perp \end{aligned}$$

where $\alpha(\gamma(e_2) \cup \gamma(e_3)) = \mathit{lub}(e_2, e_3)$. The lub function can be expressed as

$$\begin{aligned} \mathit{lub}(e_2, e_3) = & e_2 \quad \mathbf{if } e_3 = \perp \\ & e_3 \quad \mathbf{if } e_2 = \perp \\ & \mathit{all} \quad \mathbf{if } e_2 = \mathit{all} \vee e_3 = \mathit{all} \\ & e_2 \quad \mathbf{if } e_2 = e_3 \\ & \mathit{all} \quad \mathbf{if } \mathit{atom}(e_2) \vee \mathit{atom}(e_3) \\ & \tilde{c\text{ons}}(\mathit{lub}(\tilde{c\text{ar}}(e_2), \tilde{c\text{ar}}(e_3)), \mathit{lub}(\tilde{c\text{dr}}(e_2), \tilde{c\text{dr}}(e_3))) \\ & \mathbf{otherwise} \end{aligned}$$

Interpreting the step counting version in the \tilde{V}_\perp domain requires a semantics specialized to work on natural numbers. We cannot just use the above extension of function because computing the step counting version on a larger set (or rather the abstraction in \tilde{V}_\perp of this subset) should give an upper bound of the step counting version on elements of this subset. If we use the extension in \tilde{V}_\perp we will get the least upper bound of different time values, which is the symbol all . Instead we will define a special extension for functions giving numbers (the flat domain N_\perp). To this we use the domain: N_\perp^∞ of the natural numbers extended with a top element ∞ and a bottom element \perp . The integer ordering \geq is used, and $x \leq \infty, \forall x$, hence the domain N_\perp^∞ has infinite height with ∞ as a limit point. Maximal elements of subsets of N_\perp^∞ are found by the function max^∞ :

Definition 4.17. The continuous function $\mathit{max}^\infty : \mathbf{P}(N) \rightarrow N_\perp^\infty$ is defined as

$$\begin{aligned} \mathit{max}^\infty(S) = & \perp \quad \mathbf{if } S = \emptyset \\ & \mathit{max}(S) \quad \mathbf{if } S \text{ is finite} \\ & \infty \quad \mathbf{otherwise} \end{aligned}$$

where the function $\mathit{max} : \mathbf{P}(N) \rightarrow N_\perp$ gives for a set S the maximal element in S if S is finite and not empty, else the bottom element \perp .

The function max^∞ is an *abstraction* function, and we need also a *concretization* function [Cousot & Cousot, 1977].

Definition 4.18. A continuous function $\xi : N_\perp^\infty \rightarrow \mathbf{P}(N)$ is defined by

$$\begin{aligned} \xi(x) = \emptyset & \quad \text{if } x = \perp \\ N & \quad \text{if } x = \infty \\ \{0, \dots, x\} & \quad \text{otherwise} \end{aligned}$$

Lemma 4.19. The functions max^∞ and ξ satisfy

$$\begin{aligned} \xi \circ max^\infty(S) &\supseteq S & S \in \mathbf{P}(N) \\ max^\infty \circ \xi(x) &= x & x \in N_\perp^\infty \end{aligned}$$

$$N_\perp^\infty \begin{array}{c} \xrightarrow{\xi} \\ \xleftarrow{max^\infty} \end{array} \supseteq \mathbf{P}(N)$$

As before functions from V_\perp can be extended to functions from \tilde{V}_\perp , but now the definition is :

Definition 4.20. A function $f : V_\perp^k \rightarrow N_\perp$ can be extended to a function $f^\infty : \tilde{V}_\perp^k \rightarrow N_\perp^\infty$ by

$$f^\infty(x_1, \dots, x_k) = max^\infty(f_*(\gamma(x_1), \dots, \gamma(x_k)))$$

where $f_* : \mathbf{P}(V)^k \rightarrow \mathbf{P}(N)$

$$\begin{aligned} f_*(S_1, \dots, S_k) &= \{f(x_1, \dots, x_k) \mid x_i \in S_i \cup \{\perp\}, i = 1, \dots, k \\ &\quad \wedge f(x_1, \dots, x_k) \neq \perp\} \end{aligned}$$

A function $f : N_\perp^k \rightarrow N_\perp$ can be extended to a function $f^\infty : N_\perp^{\infty k} \rightarrow N_\perp^\infty$ by

$$f^\infty(x_1, \dots, x_k) = max^\infty(f_*(\xi(x_1), \dots, \xi(x_k)))$$

where $f_* : \mathbf{P}(N)^k \rightarrow \mathbf{P}(N)$

$$\begin{aligned} f_*(S_1, \dots, S_k) &= \{f(x_1, \dots, x_k) \mid x_i \in S_i \cup \{\perp\}, i = 1, \dots, k \\ &\quad \wedge f(x_1, \dots, x_k) \neq \perp\} \end{aligned}$$

Lemma 4.21. If $f : V_{\perp}^k \rightarrow N_{\perp}$ is continuous then the function $f^{\infty} : \tilde{V}_{\perp}^k \rightarrow N_{\perp}^{\infty}$ will also be continuous.

Proof. The mapping $max^{\infty} : \mathbf{P}(N) \rightarrow N_{\perp}^{\infty}$ is continuous, when $\mathbf{P}(N)$ uses set inclusion ordering. \square

Least upper bound 4.22. The least upper bound in the domain N_{\perp}^{∞} is

$$\begin{aligned} max : N_{\perp}^{\infty} \times N_{\perp}^{\infty} &\rightarrow N_{\perp}^{\infty} \\ max(x_1, x_2) &= max^{\infty}(\xi(x_1) \cup \xi(x_2)) \end{aligned}$$

Extending operations. 4.23. The continuous extension of operation 'add' and the conditional are straightforward

$$\begin{aligned} add(x, y) &= \perp && \mathbf{if} \ x = \perp \vee y = \perp \\ & \ x + y && \mathbf{otherwise} \\ add^{\infty}(x, y) &= \perp && \mathbf{if} \ x = \perp \vee y = \perp \\ & \ \infty && \mathbf{if} \ x = \infty \vee y = \infty \\ & \ x + y && \mathbf{otherwise} \end{aligned}$$

$$\begin{aligned} cond(e_1, e_2, e_3) &= \mathbf{case} \ e_1 \ \mathbf{of} \\ & \ \mathit{true} && : e_2 \\ & \ \mathit{false} && : e_3 \\ & \ \mathbf{otherwise} && \perp \\ cond^{\infty}(e_1, e_2, e_3) &= \mathit{case} \ e_1 \ \mathit{of} \\ & \ \mathit{true} && : e_2 \\ & \ \mathit{false} && : e_3 \\ & \ \mathit{all} && : max(e_2, e_3) \\ & \ \mathbf{otherwise} && \perp \end{aligned}$$

where

$$\begin{aligned} max(e_2, e_3) &= e_2 && \mathbf{if} \ e_3 = \perp \\ & \ e_3 && \mathbf{if} \ e_2 = \perp \\ & \ \infty && \mathbf{if} \ e_2 = \infty \vee e_3 = \infty \\ & \ e_2 && \mathbf{if} \ e_2 \geq e_3 \\ & \ e_3 && \mathbf{otherwise} \end{aligned}$$

The mappings $add^\infty : N_\perp^{\infty 2} \rightarrow N_\perp^\infty$ and $cond^\infty : \tilde{V}_\perp \times N_\perp^{\infty 2} \rightarrow N_\perp^\infty$ are obviously continuous.

In step counting versions we only need these two functions, but also other numeric functions can be extended in this way. As example will the subtraction function be

$$sub^\infty(x, y) = \begin{array}{l} \perp \quad \text{if } x = \perp \vee y = \perp \\ x \quad \text{otherwise} \end{array}$$

This might not be what one would expect, but it is the continuous extension to N_\perp^∞ , defined by

$$sub^\infty(x, y) = max^\infty\{sub(a, b) \mid a \in \xi(x) \wedge b \in \xi(y)\}$$

The semantics for programs on \tilde{V}_\perp can now be given.

4.4 The \tilde{V}_\perp - Semantics.

The \tilde{V}_\perp - Interpretation of the language is denoted $\tilde{\mathbf{U}}$ and the sets are :

D_σ	$= \tilde{V}_\perp$	The set of partly known S-expression
D_τ	$= N_\perp^\infty$	The natural numbers with integer ordering
Φ_σ	$= \tilde{V}_\perp^k \rightarrow \tilde{V}_\perp$	
Φ_τ	$= \tilde{V}_\perp^k \rightarrow N_\perp^\infty$	
C		- not used
R_σ	$= D_\sigma$	
R_τ	$= D_\tau$	

The auxiliary functions are

$$\begin{aligned}
atom_\tau(n) &= n && \text{if } n \in N \\
& \perp && \text{otherwise} \\
add_\tau(e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\
& \infty && \text{if some } e_i = \infty \\
& e_1 + \dots + e_k && \text{otherwise} \\
cond_\tau(e_1, e_2, e_3) &= max(e_2, e_3) && \text{if } e_1 = all \\
& e_2 && \text{if } e_1 = true \\
& e_3 && \text{if } e_1 = false \\
& \perp && \text{otherwise} \\
apply_{\tau_i}(\phi, e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\
& \phi_i(e_1, \dots, e_k) && \text{otherwise} \\
fetch_{\sigma_i}(v) &= v_i \\
atom_\sigma('a) &= 'a \\
apply_{\sigma_i}(\phi, e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\
& \phi_i(e_1, \dots, e_k) && \text{otherwise} \\
basic_{\sigma_i}(\phi, e_1, \dots, e_k) &= \perp && \text{if some } e_i = \perp \\
& \tilde{op}(e_1, \dots, e_k) && \text{otherwise} \\
cond_\sigma(e_1, e_2, e_3) &= e_2 \sqcup e_3 && \text{if } e_1 = all \\
& e_2 && \text{if } e_1 = true \\
& e_3 && \text{if } e_1 = false \\
& \perp && \text{otherwise} \\
init(\mathbf{c}, \phi, \tau) &= (\phi, \tau) \\
iterate(\phi, \tau, f_1, \dots, f_k, t_1, \dots, t_k) &= \langle f_1, \dots, f_k, t_1, \dots, t_k \rangle
\end{aligned}$$

Correctness (sketched). We want to show that the \tilde{V}_\perp - semantics $\tilde{\mathbf{U}}$ is an approximation of the collecting semantics \mathbf{U}_C , by proving :

$$\begin{aligned}
\mathbf{U}_C[[pgm]] \downarrow i(S_1, \dots, S_k) &\subseteq \xi \circ Ut[[pgm]] \downarrow i(\alpha(S_1), \dots, \alpha(S_k)) \quad i = 1, \dots, n \\
\mathbf{U}_C[[pgm]] \downarrow i(S_1, \dots, S_k) &\subseteq \gamma \circ \tilde{\mathbf{U}}[[pgm]] \downarrow i(\alpha(S_1), \dots, \alpha(S_k)) \quad i = n + 1, \dots, 2n \\
S_1, \dots, S_k &\in \mathbf{P}(V)
\end{aligned}$$

We just need to show that for $\phi \in \tilde{V}_\perp^k \rightarrow \tilde{V}_\perp$ and $\Phi \in \mathbf{P}(V)^k \rightarrow \mathbf{P}(V)$ that

$$\begin{aligned}\Phi_i(S_1, \dots, S_k) &\subseteq \gamma(\phi(\alpha(S_1), \dots, \alpha(S_k))) \text{ and} \\ \Theta_i(S_1, \dots, S_k) &\subseteq \xi(\phi(\alpha(S_1), \dots, \alpha(S_k)))\end{aligned}$$

implies

$$\begin{aligned}\mathbf{E}_{\tau C}[\langle exp \rangle] \Phi \Theta(S_1, \dots, S_k) &\subseteq \xi \circ \tilde{\mathbf{E}}_\sigma[\langle exp \rangle] \phi \tau(\alpha(S_1), \dots, \alpha(S_k)) \\ \mathbf{E}_{\sigma C}[\langle exp \rangle] \Phi \Theta(S_1, \dots, S_k) &\subseteq \gamma \circ \tilde{\mathbf{E}}_\sigma[\langle exp \rangle] \phi \tau(\alpha(S_1), \dots, \alpha(S_k))\end{aligned}$$

This is fairly easy to see by structural induction on the syntax of expressions.

Safeness Theorem 4.24. With $\tilde{\mathbf{L}}[\langle pgm \rangle] = \tilde{\mathbf{U}}[\langle pgm \rangle] \downarrow 1$ the \tilde{V}_\perp semantics satisfy

$$\tilde{\mathbf{L}}\langle pgm \rangle(X_1, \dots, X_k) \geq \max^\infty(\mathbf{L}_C\langle pgm \rangle(\gamma(X_1), \dots, \gamma(X_k))) \forall X_i \in \tilde{V}_\perp$$

and the $\tilde{\mathbf{L}}\langle pgm \rangle$ function is continuous. \square

4.5 Termination (nontermination).

The \tilde{V}_\perp – semantics interpretation is not guaranteed to terminate because of the infinite sets N_\perp^∞ and \tilde{V}_\perp domain. This is not a surprise because we cannot make a total time bound function. But we know that if the interpretation terminates, then the value will be an upper bound of the running time of the program. In other words it gives a partially correct time bound program.

There are two problems in using the \tilde{V}_\perp interpretation for a time bound program:

1. \perp is used as an explicit value in the definition of *lub* and \max^∞ .
2. the ∞ value is not a computable value, as it is a limit point for infinite chains of additions and \max^∞ .

These two problems must be solved before it is possible to compile programs from the \tilde{V}_\perp semantics to the standard semantics.

4.6 Inverted length functions.

The generation of time bound programs uses inversion of length functions. If the size of an argument to the analyzed program should be measured by a function *length* we must find a program `length-1` that fulfills this condition.

Condition 4.25.

$$\tilde{\mathbf{L}} \text{length-1}\langle n \rangle \sqsubseteq \alpha(\text{length}^{-1}(n))$$

where

$$\text{length}^{-1}(n) = \{x \in V \mid \text{length}(x) = n\}$$

The program

```
length-1(x) = 'all
```

will always satisfy this condition, but a better approximation gives better time bound functions.

Once the `length-1` is found it can be used in the algorithm to any analyzed program.

Example. 4.26. The most obvious example is to invert the `length` function from Lisp

```
length(x) = (if (eq x nil) then 0
              else (add 1 (call length (car x))))
```

that computes the function

$$\begin{aligned} \text{length}(x) = & \perp \quad \mathbf{if} \ x = \perp \\ & 0 \quad \mathbf{if} \ x = \text{nil} \\ & \perp \quad \mathbf{if} \ \text{atom}(x) \\ & 1 + \text{length}(\text{cdr}(x)) \\ & \mathbf{otherwise} \end{aligned}$$

The inverted function is

$$\begin{aligned} \text{length}^{-1}(n) = & \emptyset \quad \mathbf{if} \ n = \perp \vee n < 0 \\ & \{\text{nil}\} \quad \mathbf{if} \ n = 0 \\ & \{(a.b) \mid a \in V, b \in \text{length}^{-1}(n-1)\} \\ & \mathbf{otherwise} \end{aligned}$$

The program `length-1` should compute the function

$$\begin{aligned} \alpha(\text{length-1}(n)) = & \perp \quad \mathbf{if} \ n = \perp \vee n < 0 \\ & \text{nil} \quad \mathbf{if} \ n = 0 \\ & \text{c\~{o}ns}(\text{all}, \alpha(\text{length}^{-1}(n-1))) \end{aligned}$$

This is done by the program

```
length-1(n) = (if (eq n 0) then nil
                else (cons 'all (call length-1 (sub n 1))))
```

Example. 4.27. To specify an argument as irrelevant we cannot start with the *length* function, as no measure should be used. Instead a function $length^{-1} : N \rightarrow \mathbf{P}(V)$ can be used:

$$length^{-1}(x) = \emptyset \quad \text{if } x = \perp \\ V \quad \text{otherwise}$$

This gives

$$\alpha(length - 1(x)) = \perp \quad \text{if } x = \perp \\ all \quad \text{otherwise}$$

and the program is

```
length-1(x) = 'all
```

4.7 Program Composition.

The next theorem is a well known result from the theory of recursive functions slightly disguised for our application.

Theorem 4.21 4.28. It is possible to construct a program *tbl* from the programs t_p and *length-1* such that

$$\tilde{\mathbf{L}} \text{tbl} \langle n_1, \dots, n_k \rangle = \\ \tilde{\mathbf{L}} t_p \langle \tilde{\mathbf{L}} \text{length-1} \langle n_1 \rangle, \dots, \tilde{\mathbf{L}} \text{length-1} \langle n_k \rangle \rangle$$

Proof. If *t* is the first function in the program t_p then the program *tbl* would be

```
tb(x1, ..., xk) = (call t (call length-1 x1) ... (call length-1 x2))
```

```
t (x1, ..., xk) = ...
```

```
.
```

```
.
```

```
.
```

```
length-1 (x) = ...
```

□

4.8 Compilation

First we can summarize the construction in the theorem

Theorem 4.22 4.29. The program `tbl` in the $\tilde{\mathbf{L}}$ semantics computes a safe time bound function:

$$\begin{aligned} & \forall n_1, \dots, n_k \in N : \\ & \tilde{\mathbf{L}} \text{tbl} \langle n_1, \dots, n_k \rangle \geq \max^\infty \{ \mathbf{L} \mathfrak{t}_p \langle x_1, \dots, x_k \rangle \mid x_i \in \text{length}^{-1}(n_i) \} \end{aligned}$$

Proof. Given n_1, \dots, n_k then

$$\begin{aligned} & \tilde{\mathbf{L}} \text{tbl} \langle n_1, \dots, n_k \rangle \\ &= \tilde{\mathbf{L}} \mathfrak{t}_p \langle \tilde{\mathbf{L}} \llbracket \text{length-1} \rrbracket \langle n_1 \rangle, \dots, \tilde{\mathbf{L}} \llbracket \text{length-1} \rrbracket \langle n_k \rangle \rangle \\ &\geq \tilde{\mathbf{L}} \mathfrak{t}_p \langle \alpha(\text{length}^{-1}(n_1)), \dots, \alpha(\text{length}^{-1}(n_k)) \rangle > \\ &\geq \max^\infty \mathbf{L}_C \mathfrak{t}_p \langle \gamma(\alpha(\text{length}^{-1}(n_1))), \dots, \gamma(\alpha(\text{length}^{-1}(n_k))) \rangle > \\ &\geq \max^\infty \{ \mathbf{L} \mathfrak{t}_p \langle x_1, \dots, x_k \rangle \mid x_i \in \gamma(\alpha(\text{length}^{-1}(n_i))) \} \\ &\geq \max^\infty \{ \mathbf{L} \mathfrak{t}_p \langle x_1, \dots, x_k \rangle \mid x_i \in \text{length}^{-1}(n_i) \} \end{aligned}$$

□

The next step is to compile the `tbl` program to the standard semantics

$$\tilde{\mathbf{L}} \text{tbl} \langle n_1, \dots, n_k \rangle \rightsquigarrow \mathbf{L} \text{tb}_p \langle n_1, \dots, n_k \rangle$$

but this cannot be done in general due to the problems stated in section 4.5 - we would then require that the `tbp` program will terminate if `tp` terminates for all $x_i \in \text{length}^{-1}(n_i)$. Instead we introduce a partial correctness ordering:

Definition 4.30.

$$\forall x, y \in N_\perp : x \succeq y \stackrel{\text{def}}{\iff} x = \perp \vee x \geq y$$

The condition for the time bound program can be formulated:

Condition 4.31. `tbp` is a time bound program if

$$\mathbf{L} \text{tb}_p \langle n_1, \dots, n_k \rangle \succeq \max \{ \mathbf{L} \mathfrak{t}_p \langle n_1, \dots, n_k \rangle \mid x_i \in \text{length}^{-1}(n_i) \}$$

Compiling the \mathbf{tbl} program to a \mathbf{tb}_p program, such that

$$\mathbf{Ltb}_p\langle n_1, \dots, n_k \rangle \succeq \tilde{\mathbf{L}} \mathbf{tbl}\langle n_1, \dots, n_k \rangle$$

is done in two steps: First the $\tilde{\mathbf{L}}$ semantics is changed a little bit to $\tilde{\mathbf{L}}'$ by :

1. Using the flat domain N_\perp in stead of N_\perp^∞ .
2. Using strict versions of lub and max :

$$\begin{aligned} lub'(x_1, x_2) = & \perp && \mathbf{if} \ x_1 = \perp \vee x_2 = \perp \\ & all && \mathbf{if} \ x_1 = all \vee x_2 = all \\ & x_1 && \mathbf{if} \ x_1 = x_2 \\ & all && \mathbf{if} \ atom(x_1) \vee atom(x_2) \\ & cons(lub'(car(x_1), car(x_2)), lub'(cdr(x_1), cdr(x_2))) \\ & && \mathbf{otherwise} \end{aligned}$$

and

$$\begin{aligned} max'(x_1, x_2) = & \perp && \mathbf{if} \ x_1 = \perp \vee x_2 = \perp \\ & x_1 && \mathbf{if} \ x_1 \geq x_2 \\ & x_2 && \mathbf{otherwise} \end{aligned}$$

The element ' ∞ ' cannot appear as a constant in the program and will only be found as a limit point in the $\tilde{\mathbf{L}}$ semantics. The two points above will interpret an infinite recursion as the bottom element ' \perp ' instead of the infinity element ' ∞ '. The interpretation makes it possible to compile the program to the standard interpretation. Without detailed proof we notice that

$$\tilde{\mathbf{L}}' \mathbf{tbl}\langle n_1, \dots, n_k \rangle \succeq \tilde{\mathbf{L}} \mathbf{tbl}\langle n_1, \dots, n_k \rangle$$

The only change is in the interpretation of the \mathbf{if} expression, and it should be obvious that

$$\begin{aligned} lub'(x, y) & \succeq lub(x, y) = \alpha(\gamma(x) \cup \gamma(y)) \\ max'(x, y) & \succeq max(x, y) = max^\infty(\xi(x) \cup \xi(y)) \end{aligned}$$

To compile the program \mathbf{tbl} from the $\tilde{\mathbf{L}}'$ semantics to the standard semantics is straightforward. The $\tilde{\mathbf{L}}'$ semantics is well defined, and a compiler could be constructed by a compiler generator. The value ' all ' can be represented as the

atom `all`, and the basic operations should be compiled so they test for this value. The infinity symbol will not appear. In the optimal case the compiler should produce closed form expressions, but this requires the use of many optimizations. The example shows the result of a naive compilation.

Example. A straightforward compilation of the introductory example with the `union` function would give a very long program. The `member` function alone would be something like:

```
member (x s) = (call member1 (call eql s nil) x s)

member1 (x--12 x s) =
  (if (eq x--12 'all)
    then (call lub false
           (call member2 (call eql x (call carl s)) x s))
    else (if x--12
            then false
            else (call member2 (call eql x (call carl s)) x s)
          )
  )

member2 (x--10 x s) =
  (if (eq x--10 'all)
    then (call lub true (call member x (call cdr1 s)))
    else (if x--10
            then true
            else (call member x (call cdr1 s))
          )
  )

lub (x y) =
  (if (eq x 'all)
    then 'all
    else (if (eq y 'all)
            then 'all
            else(if (atom x)
                    then (if (eq x y) then x else 'all)
                    else (if (atom y)
                            then 'all
                            else (cons (call lub (car x) (car y))
                                         (call lub (cdr x) (cdr y))
                          )
            )
  )

eql (e1 e2) = (if (eq e1 'all)
                  then 'all
                  else (if (eq e2 'all)
                          then 'all
                          else (eq e1 e2)
                        )
                )
```

```
car1 (e1) = (if (eq e1 'all) then 'all else (car e1))
```

```
cdr1 (e1) = (if (eq e1 'all) then 'all else (cdr e1))
```

Knowing that the arguments are results of the `length-1` function makes it possible to reduce the program to

```
member (x s) = (if (eq s nil) then false else 'all)
```

The next chapters examine a way to optimize time bound functions with the objective to generate closed-form expressions when possible.

Chapter 5

Data Flow Analysis

The result from the last chapter was a time bound program `tbl` to be interpreted in the \tilde{V} -semantics. The semantics was formally defined and the program `tbl` can be straightforwardly compiled to the standard semantics. In the compiled program the '*all*' will be represented by the atom '`all`', but this means that innumerable tests for this value must be inserted around in the program. As example will the expression "`(car x)`" be translated to

```
(if (eq x 'all) then 'all else (car x))
```

Many of these tests can be decided at compile time (analysis time). Many of the tests can only have one value in the context, and using a data flow analysis that value can be found at compile time.

The flow analysis will to each function in the program compute a safe, finite description of the range and domain. The set description is a context free grammar, where the nonterminals are the function names, the terminals are the atoms appearing in the program and the constructors are 'cons' expressions. A set description may contain other function names, hence a set is only defined via the whole function environment.

The flow analysis is presented in section 5.4 and in section 5.6 an example shows its use. The other sections presents the proof, and section 5.1 5.3 and 5.5 can be skipped in a first reading. In section 5.2 we give a definition of set descriptions. In section 5.3 functions on \tilde{V}_\perp are extended to set descriptions to be used in the flow analysis. The flow analysis is proven correct by showing that it is a safe approximation of collecting semantics of \tilde{V}_\perp , presented in section 5.1.

The flow analysis is inspired by [Reynolds, 1969], and his method is here extended from the standard semantics to \tilde{V}_\perp - semantics, and the method is formalized in the abstract interpretation framework. There are many levels involved in the flow analysis as the \tilde{V}_\perp - semantics in itself is an approximation of a collecting semantics. The next figure might help the understanding. A program `p` denotes a function f in the standard semantics. The function can be extended to f_* in the collecting semantics. In last chapter we saw how to approximate this function by interpreting the program in the \tilde{V}_\perp - semantics : $\tilde{f} = \tilde{\mathbf{L}} \mathbf{p}$. Now we define a collecting semantics for the \tilde{V}_\perp - semantics, called $\tilde{\mathbf{L}}_C$, hence \tilde{f} can be extended to \tilde{f}_* with the collecting semantics.

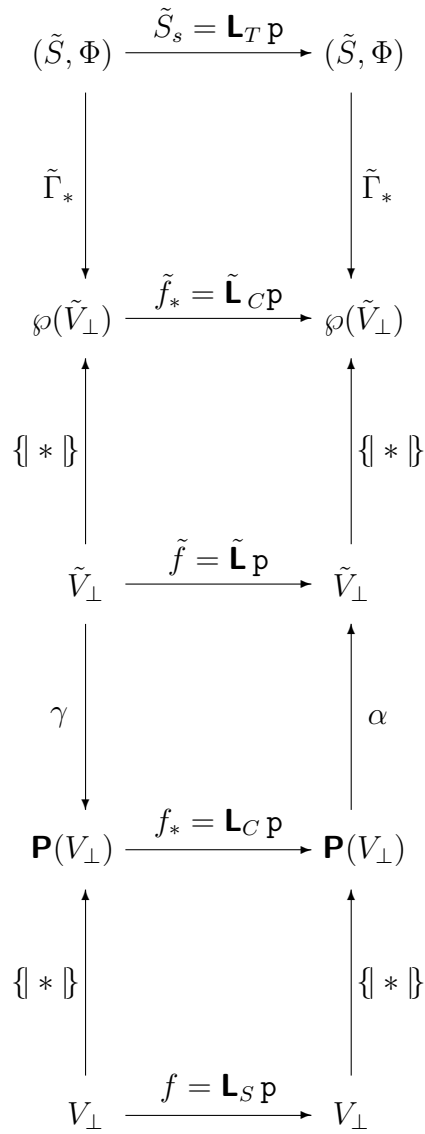


Figure 5.1: Data flow analysis

E_τ	–	as before
E_σ	\rightarrow	X_i $ $ $'a$ $ $ $(\text{cons}(\text{call } F_i E_{\sigma_1}, \dots, E_{\sigma_k}), (\text{call } F_j E_{\sigma_1}, \dots, E_{\sigma_k}))$ $ $ $(A_i E_{\sigma_1}, \dots, E_{\sigma_k})$ $ $ $(\text{call } F_i E_{\sigma_1}, \dots, E_{\sigma_k})$ $ $ $(\text{if } E_{\sigma_1} \text{ then } E_{\sigma_2} \text{ else } E_{\sigma_3})$
Pgm	–	as before

Figure 5.2: Restricted Syntax

The flow analysis to be described will be called \mathbf{L}_T , index T for terms. A set description will be in the form of a set of *terms*, and it will belong to a set called \tilde{S} . The analysis will generate set descriptions for range and domain of all functions in the program. This function environment will be of type $\Phi = (\tilde{S}^k \times \tilde{S})^n$. In section 5.2 we define a function

$$\tilde{\Gamma}_* : \tilde{S} \times \Phi \rightarrow \wp(\tilde{V}_\perp)$$

from a set description and the function environment into the powerdomain of \tilde{V}_\perp .

The analysis requires a small change in the syntax of the language: the arguments to a 'cons' operation must be function calls. Reynolds' algorithm begins by labeling the subexpressions in the program, and to each subexpression is assigned a setname. This is analogous to viewing all subexpressions as program points, but in our semantics framework only right hand sides of function definitions are program points. In a set description there will only appear set names assigned to whole right hand sides and to arguments of cons-expressions. We therefore introduce a restricted syntax for this data flow analysis (cp. section 3.4): And in the semantic framework we introduce

$$\begin{aligned} & \mathbf{E}_\sigma \llbracket (\text{cons}(\text{call } F_i E_{\sigma_1}, \dots, E_{\sigma_k}), (\text{call } F_j E_{\sigma_1}, \dots, E_{\sigma_k})) \rrbracket \theta \phi \nu \\ & = \text{cons}_\sigma(F_i, F_j, \mathbf{E}_\sigma \llbracket (\text{call } F_i E_{\sigma_1}, \dots, E_{\sigma_k}) \rrbracket \theta \phi \nu, \mathbf{E}_\sigma \llbracket (\text{call } F_j E_{\sigma_1}, \dots, E_{\sigma_k}) \rrbracket \theta \phi \nu) \end{aligned}$$

where the function $\text{cons}_\sigma : Fn \times Fn \times R_\sigma \times R_\sigma \rightarrow R_\sigma$ should be a part of the interpretation.

5.1 Collecting Semantics

In section 4.1 the Hoare powerdomain was introduced to describe the collecting semantics of the standard semantics. As the standard semantics only works with flat domains it was only necessary to use powersets, isomorphic with the Hoare powerdomain. In this section we will describe the collecting semantics of the \tilde{V}_\perp -semantics and the domains involved here will not be flat.

As in section 4.1 we use the so-called *independent attribute method* (compare [Nielson, 1984]) to the collecting interpretation. The interpretation is only used to prove that the data flow analysis in section 5.4 is a safe approximation. For this purpose this simple, less precise, but safe approximation is sufficient.

The powerdomain $\wp(\tilde{V}_\perp)$ (see section 4.1) is the set of left closures of non-empty subsets of \tilde{V}_\perp . The left closures in $\wp(\tilde{V}_\perp)$ will also be Scott-closed because of the finite height of \tilde{V}_\perp . A left closed set containing an element with an *all* symbol somewhere in a structure will also contain elements with anything else at that place in the structure. Especially a set containing the symbol *all* will also contain the whole \tilde{V}_\perp . It should be obvious that the basic operations $c\tilde{o}n_s$, $c\tilde{a}r$, and $c\tilde{d}r$ are stable with respect to this powerdomain, when they are extended to subsets by

$$\begin{aligned} c\tilde{o}n_{s*}(x, y) &= \{c\tilde{o}n_s(a, b) \mid a \in x \wedge b \in y\} && : \wp(\tilde{V}_\perp) \times \wp(\tilde{V}_\perp) \rightarrow \wp(\tilde{V}_\perp) \\ c\tilde{a}r_*(x) &= \{c\tilde{a}r(a) \mid a \in x\} && : \wp(\tilde{V}_\perp) \rightarrow \wp(\tilde{V}_\perp) \\ c\tilde{d}r_*(x) &= \{c\tilde{d}r(a) \mid a \in x\} && : \wp(\tilde{V}_\perp) \rightarrow \wp(\tilde{V}_\perp) \end{aligned}$$

The powerdomain $\wp(N_\perp^\infty)$ is isomorphic with N_\perp^∞ . All finite subsets not containing the infinite element ∞ will be totally described by their maximal element. There is only one infinite set in $\wp(N_\perp^\infty)$ namely N_\perp^∞ . The elements in $\wp(N_\perp^\infty)$ are closed under union and element-wise addition. The Scott closed subsets of N_\perp^∞ can be found by $\mathbf{SC} = \{\} * \} \circ max^\infty$ or as

$$\mathbf{SC}(X) = \mathbf{LC}(\{max^\infty(X)\})$$

Functions $\tilde{f} : \tilde{V}_\perp^k \rightarrow \tilde{V}_\perp$ and $\tilde{t} : \tilde{V}_\perp^k \rightarrow N_\perp^\infty$ can be extended to continuous functions $\tilde{F} : \wp(\tilde{V}_\perp)^k \rightarrow \wp(\tilde{V}_\perp)$ and $\tilde{T} : \wp(\tilde{V}_\perp)^k \rightarrow \wp(N_\perp^\infty)$ by

$$\begin{aligned} \tilde{F}(X_1, \dots, X_k) &= \mathbf{LC}(\{f(x_1, \dots, x_k) \mid x_i \in X_i\}) \\ \tilde{T}(X_1, \dots, X_k) &= \{y \in N_\perp^\infty \mid y \leq max^\infty(\{t(x_1, \dots, x_k) \mid x_i \in X_i\})\} \end{aligned}$$

We can define the function $\mathbf{LC} : \mathbf{P}(\tilde{V}_\perp) \rightarrow \wp(\tilde{V}_\perp)$ formally as

$$\mathbf{LC}(X) = \bigcup_{x \in X} \mathbf{LC}_1(x)$$

$$\begin{aligned}
\mathbf{LC}_1(x) = & \tilde{V}_\perp && \text{if } x = all && , \text{ else} \\
& x && \text{if } atom(x) && , \text{ else} \\
& \{c\tilde{o}n s(x_1, x_2) \mid x_1 \in \mathbf{LC}_1(a) \wedge x_2 \in \mathbf{LC}_1(b)\} \\
& && \text{if } x = (a.b)
\end{aligned}$$

Collecting semantics

The Collecting Interpretation of the \tilde{V}_\perp semantics is denoted $\tilde{\mathbf{U}}_C$ and the sets are:

$$\begin{aligned}
D_\sigma &= \wp(\tilde{V}_\perp) && \text{The powerdomain of S-expression with } all \text{ symbols} \\
D_\tau &= \wp(N_\perp^\infty) && \text{The powerdomain of natural numbers with } \infty \\
\Phi_\sigma &= \wp(\tilde{V}_\perp)^k \rightarrow \wp(\tilde{V}_\perp) \\
\Phi_\tau &= \wp(\tilde{V}_\perp)^k \rightarrow \wp(N_\perp^\infty) \\
C &&& - \text{ not used} \\
R_\sigma &= D_\sigma \\
R_\tau &= D_\tau
\end{aligned}$$

The auxiliary functions are

$$\begin{aligned}
atom_\tau(n) &= \{0, \dots, n\} \cup \{\perp\} && \text{if } n \in N \\
& && \{\perp\} && \text{otherwise} \\
add_\tau(e_1, \dots, e_k) &= \{\perp\} && \text{if some } e_i = \{\perp\}, \text{ else} \\
& && N_\perp^\infty && \text{if some } e_i = \infty, \text{ else} \\
& && \{x_1 + \dots + x_k \mid x_i \in e_i \setminus \{\perp\}\} \cup \{\perp\}
\end{aligned}$$

$$cond_\tau(e_1, e_2, e_3) = e_t \cup e_f \cup e_a$$

where

$$e_t = \text{if } true \in e_1 \text{ then } e_2 \text{ else } \{\perp\}$$

$$e_f = \text{if } false \in e_1 \text{ then } e_3 \text{ else } \{\perp\}$$

$$e_a = \text{if } all \in e_1 \text{ then } \mathbf{LC}\{max(x_2, x_3) \mid x_2 \in e_2 \wedge x_3 \in e_3\} \text{ else } \{\perp\}$$

$$\begin{aligned}
apply_{\tau_i}(\theta, e_1, \dots, e_k) &= \{\perp\} && \text{if some } e_j = \{\perp\} \\
& && \theta_i(e_1, \dots, e_k) && \text{otherwise}
\end{aligned}$$

$$fetch_{\sigma_i}(\nu) = \nu_i$$

$$atom_\sigma(a) = \{a\} \cup \{\perp\}$$

$$\begin{aligned}
apply_{\sigma_i}(\phi, e_1, \dots, e_k) &= \{\perp\} && \text{if some } e_j = \{\perp\} \\
& && \phi_i(e_1, \dots, e_k) && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
basic_{\sigma_i}(\phi, e_1, \dots, e_k) &= \{\perp\} && \text{if some } e_j = \{\perp\} \\
& && \{\tilde{o}p_i(x_1, \dots, x_k) \mid x_j \in e_j\} && \text{otherwise}
\end{aligned}$$

$$cond_\sigma(e_1, e_2, e_3) = e_t \cup e_f \cup e_a$$

where

$$e_t = \text{if } true \in e_1 \text{ then } e_2 \text{ else } \{\perp\}$$

$$e_f = \text{if } false \in e_1 \text{ then } e_3 \text{ else } \{\perp\}$$

$$e_a = \text{if } all \in e_1 \text{ then } \mathbf{LC}\{x_2 \sqcup x_3 \mid x_2 \in e_2 \wedge x_3 \in e_3\} \text{ else } \{\perp\}$$

$$init(\mathbf{c}, \theta, \phi) = (\theta, \phi)$$

$$iterate(\theta, \phi, t_1, \dots, t_k, f_1, \dots, f_k) = \langle t_1, \dots, t_k, f_1, \dots, f_k \rangle$$

Correctness

We want to show that the collecting semantics $\tilde{\mathbf{U}}_C$ is an extension of the \tilde{V}_\perp - semantics $\tilde{\mathbf{U}}$, by proving :

$$\tilde{\mathbf{U}}_C[[pgm]] \downarrow i \circ \{\ast\}^k = \{\ast\} \circ \tilde{\mathbf{U}}[[pgm]] \downarrow i \quad i = 1, \dots, 2n$$

Or in other words

$$\begin{aligned} \forall i = 1, \dots, 2n, \forall x_1, \dots, x_k \in \tilde{V}_\perp \\ \tilde{\mathbf{U}}_C[[pgm]] \downarrow i (\{x_1\}, \dots, \{x_k\}) = \{\tilde{\mathbf{U}}[[pgm]] \downarrow i (x_1, \dots, x_k)\} \end{aligned}$$

We just need to show that for

$$\begin{aligned} \phi &\in (\tilde{V}_\perp^k \rightarrow \tilde{V}_\perp)^n \\ \Phi &\in (\wp(\tilde{V}_\perp)^k \rightarrow \wp(\tilde{V}_\perp))^n \\ \theta &\in (\tilde{V}_\perp^k \rightarrow N_\perp^\infty)^n \\ \Theta &\in (\wp(\tilde{V}_\perp^k) \rightarrow \wp(N_\perp^\infty))^n \end{aligned}$$

that

$$\begin{aligned} \{\ast\} \circ \phi_i &= \Phi_i \circ \{\ast\}^k \quad \text{and} \\ \{\ast\} \circ \theta_i &= \Theta_i \circ \{\ast\}^k \end{aligned}$$

implies

$$\{\tilde{\mathbf{E}}_\sigma[[\langle exp \rangle]] \theta \phi(x_1, \dots, x_k)\} = \tilde{\mathbf{E}}_{\sigma C}[[\langle exp \rangle]] \Phi \Theta(\{x_1\}, \dots, \{x_k\})$$

and

$$\{\tilde{\mathbf{E}}_\tau[[\langle exp \rangle]] \theta \phi(x_1, \dots, x_k)\} = \tilde{\mathbf{E}}_{\tau C}[[\langle exp \rangle]] \Phi \Theta(\{x_1\}, \dots, \{x_k\})$$

This can be proved by structural induction after the syntax of the expressions. We will only sketch the proof here :

The induction start is :

$$\begin{aligned} \text{atom}_\sigma : \quad \{a\} \cup \{\perp\} &= \{a\} \\ \text{fetch}_\sigma : \quad \{x_1\} &= \{x_1\} \end{aligned}$$

In the induction step we assume that

$$\begin{aligned} e_i &= \tilde{\mathbf{E}}_{\sigma C}[[E_i]] \Phi \Theta(\{x_1\}, \dots, \{x_k\}) \\ &= \{\tilde{\mathbf{E}}_\sigma[[E_i]] \theta \phi(x_1, \dots, x_k)\} \\ &= \{v_1\} \end{aligned}$$

This gives :

$$\begin{aligned}
& \{\tilde{o}p(x_1, \dots, x_k) \mid x_i \in e_i\} \\
&= \{\tilde{o}p(x_1, \dots, x_k) \mid x_i \in \{v_i\}\} \\
&= \tilde{o}p_*(\{v_1\}, \dots, \{v_k\}) \\
&= \{\tilde{o}p(v_1, \dots, v_k)\}
\end{aligned}$$

due to the continuity of $\tilde{o}p$, $\tilde{o}p_*$ and $\{\ast\}$.

For the conditional we get

$$\begin{aligned}
& e_t \cup e_f \cup e_a \\
&= (\mathbf{if} \text{ true} \in \{v_1\} \mathbf{then} \{v_2\} \mathbf{else} \{\perp\}) \cup \\
&\quad (\mathbf{if} \text{ false} \in \{v_1\} \mathbf{then} \{v_3\} \mathbf{else} \{\perp\}) \cup \\
&\quad (\mathbf{if} \text{ all} \in \{v_1\} \mathbf{then} \mathbf{LC}\{x_2 \sqcup x_3 \mid x_2 \in \{v_2\} \wedge x_3 \in \{v_3\}\} \mathbf{else} \{\perp\}) \\
&= (\mathbf{if} \text{ true} \in \{v_1\} \mathbf{then} \{v_2\} \mathbf{else} \{\perp\}) \cup \\
&\quad (\mathbf{if} \text{ false} \in \{v_1\} \mathbf{then} \{v_3\} \mathbf{else} \{\perp\}) \text{ cup} \\
&\quad (\mathbf{if} \text{ all} \in \{v_1\} \mathbf{then} \mathbf{LC}(\{v_2 \sqcup v_3\}) \mathbf{else} \{\perp\}) \\
&= \mathbf{LC}(\{\tilde{c}ond(x_1, x_2, x_3) \mid x_1 \in \{v_1\} \wedge x_2 \in \{v_2\} \wedge x_3 \in \{v_3\}\}) \\
&= \{\tilde{c}ond(v_1, v_2, v_3)\}
\end{aligned}$$

The induction step for the other constructors are secured in the same way. We have now proved that the interpretation is safe in the following sense:

Safeness Theorem 5.1. With

$$\mathbf{Ltilde}_C \langle pgm \rangle = \tilde{\mathbf{U}}_C \llbracket \langle pgm \rangle \rrbracket \downarrow 1$$

the collecting interpretation satisfies

$$\{\tilde{\mathbf{L}} \langle pgm \rangle(x_1, \dots, x_k)\} = \mathbf{Ltilde}_C \langle pgm \rangle(\{x_1\}, \dots, \{x_k\}) \quad \forall x_i \in V_\perp$$

and $\mathbf{Ltilde}_C \langle pgm \rangle$ is continuous.

Corollary 5.2.

$$\begin{aligned}
& \mathbf{Ltilde}_C \langle pgm \rangle(X_1, \dots, X_k) \supseteq \{\tilde{\mathbf{L}} \langle pgm \rangle(x_1, \dots, x_k) \mid x_i \in X_i\} \\
& \quad \forall X_i \in \wp(\tilde{V}_\perp)
\end{aligned}$$

5.2 The Domain of Sets of Partly Known Structures

The idea of the Reynolds' paper [Reynolds, 1969] is to generate a system of recursive set definitions, that to each subexpression assigns the set of values it can evaluate to. In the terminology of flow analysis [Jones & Muchnick, 1979] this is a forward flow analysis where each subexpression is normally considered to be a program point. In abstract interpretation of applicative programs (eg. [Jones & Mycroft, 1986]) only right hand sides of function definitions are considered as program points, but using the restricted syntax described above this gives no real difference.

Reynolds associates a set name to each subexpression and defines the set by an expression build up as a union of simple terms, where a term can be one of two. It can be a constant (atomic) denoting the singleton set of this constant, or it can be a cons expression with two set names as arguments, denoting the set of pairs with elements from each of the two sets (like a cartesian product). The constants should be atoms appearing under a quote in the program (or self-quoting constants). The set of terms to build up set definitions from are therefore finite, and this secures the termination of the analysis.

The system of set definitions is a context-free grammar, and the set description is finite even though the sets described might be infinite. This is mainly because that even though the language is eager, the description of the cons operation is lazy.

In the \tilde{V} -semantics there are new ways to produce values, and the set description must be extended to deal with the new operations in the language. There are five ways an expression can produce a value:

1. cons operation
2. car and cdr operation
3. equality and other predicates
4. number operations
5. lubs of branches in `if` expressions.

The first two points are known from the work by Reynolds. The cons term should now take two function names to denote the set of pairs with values from each of these two functions value set.

The third is new, because a test has now three possible values *true*, *false*, and *all*. In Reynolds work all tests are expected to be either *true* or *false*. Here it means that the terms to build up set definitions of should at least contain the three constants *true*, *false*, and *all*. As long as we only have finitely many terms the termination of the flow analysis is easy to prove.

Number operations on numeric constants cannot just evaluate new constants, because this will give an infinite set of terms - the natural numbers. In stead the symbol “num” is introduced to denote the set of natural numbers.

Lubs of branches can happen for both sorts ($cond_\tau$ and $cond_\sigma$ expressions) but with numbers it is just the maximum - a number operation. Lubs of sets of S-expressions is difficult to describe in a finite way. The problem is here solved by using the same type of trick as with the cons in the Reynolds paper: it is done lazily. One could do it by introducing a *termlub* that takes an arbitrary number of function names, and denotes the set of lubs with arguments from each of these functions value set. It turns out that there is an easier way to describe a lub of sets, namely by letting the cons term take two sets of function names, where these two sets should be interpreted as lubs as before. This needs a more precise definition, but informally it means that the evaluation of lubs is moved down one level in structures.

Definition 5.3. Given a program p and let the set of function names be Fn and the set of constants be $Atoms$. The set of terms will be called \tilde{T}

$$\tilde{T} = Atoms \cup \{true, false, all\} \cup \{ 'num' \} \cup \{ 'cons' \} \times \mathbf{P}(Fn) \times \mathbf{P}(Fn)$$

The powerset will be called \tilde{S}

$$\tilde{S} = \mathbf{P}(\tilde{T})$$

The function environment will give descriptions of domain and range of the program functions

$$\Phi = (\tilde{S}^k \times \tilde{S})^{2n}$$

Definition 5.4. The function $\tilde{\Gamma}_*$ gives the correspondence between \tilde{S} and $\wp(\tilde{V}_\perp)$:

$$\tilde{\Gamma}_* : \tilde{S} \times \Phi \rightarrow \wp(\tilde{V}_\perp)$$

$$\tilde{\Gamma}_*(x, \phi) = \bigcup_{t \in x} \tilde{\Gamma}(t, \phi) \cup \{\perp\}$$

$$\tilde{\Gamma}(t, \phi) = \{t\} \cup \{\perp\} \quad \text{if } atom(t)$$

$$N_\perp \quad \text{if } t = 'num'$$

$$\tilde{V}_\perp \quad \text{if } t = all$$

$$cons_*(K(G_1, \phi), K(G_2, \phi))$$

$$\text{if } t = cons(G_1, G_2)$$

where

$$K(\{g_1, \dots, g_n\}, \phi) = \text{lub}_*(\tilde{\Gamma}_*(\phi_{g_1}, \phi), \dots, \tilde{\Gamma}_*(\phi_{g_n}, \phi))$$

and

$$\text{lub}_*(x_1, \dots, x_k) = \mathbf{LC}(\{\text{lub}(e_1, \dots, e_k) \mid e_i \in x_i\})$$

A similar, but a much simpler function can establish the relation between \tilde{S} and $\wp(N_\perp^\infty)$. This function maps the term 'num' to N_\perp^∞ and it is undefined for cons-expressions and it doesn't depend on ϕ .

5.3 Term versions

Definition 5.5. A function $\tilde{f}_s : \tilde{S}^k \times \Phi \rightarrow \tilde{S}$ is called a *term version* of a function $f_* : \wp(\tilde{V}_\perp)^k \rightarrow \wp(\tilde{V}_\perp)$ if

$$\tilde{\Gamma}_*(\tilde{f}_s(x_1, \dots, x_k, \phi), \phi) \supseteq \tilde{f}_*(\tilde{\Gamma}_*(x_1, \phi), \dots, \tilde{\Gamma}_*(x_k, \phi)) \quad , \forall x_1, \dots, x_k, \phi$$

Notice that a term version will always exist: The function mapping all values to the symbol "all".

In the same way we can define term versions of function on N_\perp^∞ , but in most cases the most general function :

$$\tilde{t}_s(x_1, \dots, x_k, \phi) = \{\text{'num'}\}$$

will suffice. In the rest of this section we will describe term versions \tilde{f}_s of the basic operations (or rather the subset extended, \tilde{V} extended basic operations). First we define some simple functions and show some identities. The rest of this section can be skipped at a first reading. Especially the proofs are not important to the understanding.

Lemma 5.6. With the definition

$$\begin{aligned} \text{lub}_* : \wp(\tilde{V}_\perp)^k &\rightarrow \wp(\tilde{V}_\perp) \\ \text{lub}_*(x_1, \dots, x_k) &= \mathbf{LC}(\{\text{lub}(y_1, \dots, y_k) \mid y_i \in x_i, \forall i\}) \end{aligned}$$

We have the identities:

- a. $\text{lub}_*(\text{lub}_*(x, y), z) = \text{lub}_*(x, y, z)$
- b. $\text{lub}_*(\text{cõns}_*(x, y), \text{cõns}_*(z, v)) = \text{cõns}_*(\text{lub}_*(x, z), \text{lub}_*(y, v))$
- c. $\text{lub}_*(\text{lub}_*(x, y), \text{lub}_*(x, y)) = \text{lub}_*(x, y)$

Proof. The proofs will use the associativity and monotonicity of the lub function (see section 4.2 lemma 5).

- a. $lub_*(x, y, z)$
 $= \{lub(a, b, c) \mid a \in x \wedge b \in y \wedge c \in z\}$
 $= \mathbf{LC}(\{lub(lub(a, b), c) \mid a \in x \wedge b \in y \wedge c \in z\})$
 $= \mathbf{LC}(\{lub(d, c) \mid d = lub(a, b) \wedge a \in x \wedge b \in y \wedge c \in z\})$
 $= \mathbf{LC}(\{lub(d, c) \mid d \in \{lub(a, b) \mid a \in x \wedge b \in y\} \wedge c \in z\})$
 $= \mathbf{LC}(\{lub(d, c) \mid d \in lub_*(x, y) \wedge c \in z\})$
 $= lub_*(lub_*(x, y), z)$
- b. $lub_*(c\tilde{o}n s_*(x, y), c\tilde{o}n s_*(z, v))$
 $= \mathbf{LC}(\{lub(c\tilde{o}n s(a, b), c\tilde{o}n s(c, d)) \mid a \in x \wedge b \in y \wedge c \in z \wedge d \in v\})$
 $= \mathbf{LC}(\{c\tilde{o}n s(lub(a, c), lub(b, d)) \mid a \in x \wedge b \in y \wedge c \in z \wedge d \in v\})$
 $= c\tilde{o}n s_*(lub_*(x, z), lub_*(y, v))$
- c. There are two ways to prove :
1. $lub_*(lub_*(x, y), lub_*(x, y)) \supseteq lub_*(x, y)$

This is just a special case of the more general formula :

$$lub_*(x, x) \supseteq x$$

but notice that the equality will not in general be true. The other way says:

2. $lub_*(lub_*(x, y), lub_*(x, y)) \subseteq lub_*(x, y)$
 $\Leftrightarrow lub_*(lub_*(x, x), lub_*(y, y)) \subseteq lub_*(x, y)$
 $\Leftrightarrow lub(a, b) \in lub_*(x, y) \quad \forall a \in lub_*(x, x) \wedge b \in lub_*(y, y)$

The proof will be by structural induction after x and y :

- i. if $a = all \vee b = all$ then will either x or y contain the symbol *all* or two different atoms, and therefore will $all \in lub_*(x, y)$.
- ii. if $atom(a) \wedge atom(b)$ then will either a and b be equal and hence will both x and y contain this atom, or they will be different and $lub_*(x, y)$ will contain the symbol *all*.

- iii. if $atom(a) \vee atom(b)$ then will $all \in lub_*(x, y)$.
- iv. if $a = cons(p, q) \wedge b = cons(r, s)$ then assume the identity has been proven for $c\tilde{a}r_*(x)$, $c\tilde{a}r_*(y)$, $c\tilde{d}r_*(x)$, and $c\tilde{d}r_*(y)$ then will

$$a \in lub_*(c\tilde{o}n s_*(c\tilde{a}r_*(x), c\tilde{d}r_*(x)), c\tilde{o}n s_*(c\tilde{a}r_*(x), c\tilde{d}r_*(x)))$$

$$b \in lub_*(c\tilde{o}n s_*(c\tilde{a}r_*(y), c\tilde{d}r_*(y)), c\tilde{o}n s_*(c\tilde{a}r_*(y), c\tilde{d}r_*(y)))$$

and

$$lub(a, b) \in$$

$$lub_* (lub_*(c\tilde{o}n s_*(c\tilde{a}r_*(x), c\tilde{d}r_*(x)), c\tilde{o}n s_*(c\tilde{a}r_*(x), c\tilde{d}r_*(x))),$$

$$lub_*(c\tilde{o}n s_*(c\tilde{a}r_*(y), c\tilde{d}r_*(y)), c\tilde{o}n s_*(c\tilde{a}r_*(y), c\tilde{d}r_*(y))))$$

$$= c\tilde{o}n s_*(lub_*(lub_*(c\tilde{a}r_*(x), c\tilde{a}r_*(x)), lub_*(c\tilde{a}r_*(y), c\tilde{a}r_*(y))),$$

$$lub_*(lub_*(c\tilde{d}r_*(x), c\tilde{d}r_*(x)), lub_*(c\tilde{d}r_*(y), c\tilde{d}r_*(y))))$$

$$= c\tilde{o}n s_*(lub_*(c\tilde{a}r_*(x), c\tilde{a}r_*(y)),$$

$$lub_*(c\tilde{d}r_*(x), c\tilde{d}r_*(y)))$$

$$= lub_*(c\tilde{o}n s_*(c\tilde{a}r_*(x), c\tilde{d}r_*(x)),$$

$$c\tilde{o}n s_*(c\tilde{a}r_*(y), c\tilde{d}r_*(y)))$$

$$\subseteq lub_*(x, y)$$

And now we are ready to define the term versions of the basic operations.

Lemma 5.7. $lub_s : \tilde{S}^k \times \Phi \rightarrow \tilde{S}$ is the term version of lub_* : $\wp(\tilde{V}_\perp)^k \rightarrow \wp(\tilde{V}_\perp)$ where

$$lub_s(x_1, \dots, x_k, \phi) = \{lub_t(t_1, \dots, t_k, \phi) \mid t_i \in x_i, \forall i\}$$

$$lub_t : \tilde{T}^k \times \Phi \rightarrow \tilde{T}$$

$$lub_t(t_1, \dots, t_k, \phi) = \begin{array}{ll} all & \text{if some } t_i = num \\ t_1 & \text{if } t_1 = \dots = t_k \\ cons(\bigcup_i G_{ia}, \bigcup_i G_{id}) & \\ & \text{if } t_i = cons(G_{ia}, G_{id}), \forall i \\ all & \text{otherwise} \end{array}$$

Proof. We will only prove it for $k = 2$, and due to the monotonicity of lub_s , $\tilde{\Gamma}_*$, and lub_* is it only necessary to prove that

$$\tilde{\Gamma}(lub_t(t_1, t_2, \phi), \phi) \supseteq lub_*(\tilde{\Gamma}(t_1, \phi), \tilde{\Gamma}(t_2, \phi)) \quad , \forall t_1, t_2, \phi$$

The proof will be a case study of t_1 and t_2 :

1. If $t_1 = num$ or $t_2 = num$ then let us assume that $t_1 = num$ and we should prove that

$$\tilde{V} \supseteq lub_*(N, \tilde{\Gamma}(t_2, \phi))$$

that obviously is true.

2. If $t_1 = t_2$ then we shall prove that

$$\tilde{\Gamma}(t_1, \phi) \supseteq lub_*(\tilde{\Gamma}(t_1, \phi), \tilde{\Gamma}(t_1, \phi))$$

and there are three situations:

i.: $atom(t_1)$

$$\{t_1, \perp\} \supseteq lub_* (\{t_1, \perp\}, \{t_1, \perp\}) = \{t_1, \perp\}$$

ii.: $t_1 = all$

$$\tilde{V}_\perp \supseteq lub_*(\tilde{V}_\perp, \tilde{V}_\perp) = \tilde{V}_\perp$$

iii.: $t_1 = cons(G_1, G_2)$

$$\begin{aligned} c\tilde{o}n s_*(K(G_1, \phi), K(G_2, \phi)) &\supseteq lub_*(c\tilde{o}n s_*(K(G_1, \phi), K(G_2, \phi)), \\ &\quad c\tilde{o}n s_*(K(G_1, \phi), K(G_2, \phi))) \\ &= c\tilde{o}n s_*(lub_*(K(G_1, \phi), K(G_1, \phi)), \\ &\quad lub_*(K(G_2, \phi), K(G_2, \phi))) \end{aligned}$$

but the K function produces results via the lub_* function and we can therefore use lemma 5.5.c to get

$$\begin{aligned} &c\tilde{o}n s_*(lub_*(K(G_1, \phi), K(G_1, \phi)), \\ &\quad lub_*(K(G_2, \phi), K(G_2, \phi))) \\ &= c\tilde{o}n s_*(K(G_1, \phi), K(G_2, \phi)) \end{aligned}$$

3. If $t_1 = \text{cons}(G_1, G_2)$ and $t_2 = \text{cons}(G_3, G_4)$ then the lemma says

$$\begin{aligned} & \tilde{\Gamma}_*(\text{cons}(G_1 \cup G_3, G_2 \cup G_4), \phi) \\ & \supseteq \text{lub}_*(\text{c\~{o}n}s_*(K(G_1, \phi), K(G_2, \phi)), \text{c\~{o}n}s_*(K(G_3, \phi), K(G_4, \phi))) \\ & = \text{c\~{o}n}s_*(\text{lub}_*(K(G_1, \phi), K(G_3, \phi)), \text{lub}_*(K(G_2, \phi), K(G_4, \phi))) \end{aligned}$$

And this is true because

$$\text{lub}_*(K(G_1, \phi), K(G_2, \phi)) = K(G_1 \cup G_2, \phi)$$

4. Otherwise :

$$\tilde{\Gamma}(\text{all}, \phi) \supseteq \text{lub}_*(\tilde{\Gamma}(t_1, \phi), \tilde{\Gamma}(t_2, \phi))$$

and

$$\tilde{\Gamma}(\text{all}, \phi) = \tilde{V}$$

so it must be true.

Lemma 5.8. The function $\text{c\~{a}r}_s$ is the term version of car_* where

$$\text{c\~{a}r}_s(x, \phi) = \bigcup_{t \in x} \text{c\~{a}r}_t(t, \phi)$$

$$\begin{aligned} \text{c\~{a}r}_t(t, \phi) = & H(G_1, \phi) & \mathbf{if} \ t = \text{cons}(G_1, G_2) \\ & \{\text{all}\} & \mathbf{if} \ t = \text{all} \\ & \{\perp\} & \mathbf{otherwise} \end{aligned}$$

$$H(\{g_1, \dots, g_n\}, \phi) = \text{lub}_s(\phi_{g_1}, \dots, \phi_{g_n}, \phi)$$

Proof. We shall prove that

$$\tilde{\Gamma}_*(\text{c\~{a}r}_s(x, \phi), \phi) \supseteq \text{c\~{a}r}_*(\tilde{\Gamma}_*(x, \phi)) \quad \forall x, \phi$$

and due to the monotonicity of $\text{c\~{a}r}_s$ and $\tilde{\Gamma}_*$ we just need to prove that

$$\tilde{\Gamma}_*(\text{c\~{a}r}_t(t, \phi), \phi) \supseteq \text{c\~{a}r}_*(\tilde{\Gamma}(t, \phi)) \quad \forall t \in \tilde{T}, \phi$$

A case study gives

1. if $t = cons(G_1, G_2)$ then

$$\begin{aligned}
\tilde{c}\tilde{a}r_*(\tilde{\Gamma}(t, \phi)) &= \tilde{c}\tilde{a}r_*(\tilde{c}\tilde{o}\tilde{n}s_*(K(G_1, \phi), K(G_2, \phi))) \\
&\subseteq K(G_1, \phi) = K(\{g_1, \dots, g_k\}, \phi) \\
&= \text{lub}_*(\tilde{\Gamma}_*(\phi_{g_1}, \phi), \dots, \tilde{\Gamma}_*(\phi_{g_k}, \phi)) \\
&\subseteq \tilde{\Gamma}_*(\text{lub}_s(\phi_{g_1}, \dots, \phi_{g_k}, \phi)) \\
&= \tilde{\Gamma}_*(H(G_1, \phi)) \\
&= \tilde{\Gamma}_*(\tilde{c}\tilde{a}r_t(t, \phi), \phi)
\end{aligned}$$

2. If $t = all$ then

$$\tilde{c}\tilde{a}r_*(\tilde{\Gamma}(t, \phi)) = \tilde{c}\tilde{a}r_*(\tilde{V}_\perp) = \tilde{V}_\perp$$

and

$$\tilde{\Gamma}_*(\tilde{c}\tilde{a}r_t(t, \phi), \phi) = \tilde{\Gamma}(all, \phi) = \tilde{V}_\perp$$

3. If $t = atom(t)$ then

$$\tilde{c}\tilde{a}r_*(\tilde{\Gamma}(t, \phi)) = \tilde{c}\tilde{a}r_*(\{t, \perp\}) = \{\perp\}$$

and

$$\tilde{\Gamma}_*(\tilde{c}\tilde{a}r_t(t, \phi), \phi) = \tilde{\Gamma}_*(\emptyset, \phi) = \{\perp\}$$

Lemma 5.9. The function $\tilde{c}\tilde{o}\tilde{n}s_s : Fn \times Fn \times \Phi \rightarrow \tilde{S}$ defined as

$$\tilde{c}\tilde{o}\tilde{n}s_s(f_1, f_2, \phi) = \mathbf{if} \phi_{f_1} \neq \emptyset \wedge \phi_{f_2} \neq \emptyset \mathbf{then} \{cons(f_1, f_2)\} \mathbf{else} \emptyset$$

is a term version of $\tilde{c}\tilde{o}\tilde{n}s_*$ in the following sense

$$\tilde{\Gamma}_*(\tilde{c}\tilde{o}\tilde{n}s_s(f_1, f_2, \phi), \phi) \supseteq \tilde{c}\tilde{o}\tilde{n}s_*(\tilde{\Gamma}(\phi_{f_1}, \phi), \tilde{\Gamma}(\phi_{f_2}, \phi))$$

Proof. If $\phi_{f_1} = \emptyset$ or $\phi_{f_2} = \emptyset$ then both sides will be equal to \emptyset . Else we have

$$\begin{aligned}
\tilde{\Gamma}_*(\tilde{c}\tilde{o}\tilde{n}s_s(f_1, f_2, \phi), \phi) &= \tilde{\Gamma}(cons(f_1, f_2), \phi) \\
&= \tilde{c}\tilde{o}\tilde{n}s_*(\text{lub}_*(\tilde{\Gamma}_*(\phi_{f_1}, \phi), \text{lub}_*(\tilde{\Gamma}_*(\phi_{f_2}, \phi)))) \\
&= \tilde{c}\tilde{o}\tilde{n}s_*(\tilde{\Gamma}_*(\phi_{f_1}, \phi), \tilde{\Gamma}_*(\phi_{f_2}, \phi))
\end{aligned}$$

Lemma 5.10. The function $\tilde{e}q_s : \tilde{S} \times \tilde{S} \times \Phi \rightarrow \tilde{S}$ is a term version of eq_* , where

$$\begin{aligned} \tilde{e}q_s(x_1, x_2, \phi) &= \bigcup_{t_1 \in x_1 \wedge t_2 \in x_2} \tilde{e}q_t(t_1, t_2, \phi) \\ \tilde{e}q_t(t_1, t_2, \phi) &= \begin{array}{ll} \{all\} & \text{if } t_1 = all \vee t_2 = all \\ \{true, false\} & \text{if } t_1 = num \vee t_2 = num \\ \{false\} & \text{if } t_1 \neq t_2 \wedge atom(t_1) \wedge atom(t_2) \\ \{true\} & \text{if } t_1 = t_2 \wedge atom(t_1) \\ \emptyset & \text{otherwise} \end{array} \end{aligned}$$

Proof. We shall prove that

$$\begin{aligned} &\tilde{\Gamma}_*(\tilde{e}q_s(x_1, x_2, \phi), \phi) \supseteq \tilde{e}q_*(\tilde{\Gamma}_*(x_1, \phi), \tilde{\Gamma}_*(x_2, \phi)) \quad \forall x_1, x_2, \phi \\ &\Downarrow \\ &\bigcup_{t_1 \in x_1 \wedge t_2 \in x_2} \tilde{\Gamma}_*(\tilde{e}q_t(t_1, t_2, \phi), \phi) \supseteq \\ &\bigcup_{t_1 \in x_1 \wedge t_2 \in x_2} \mathbf{LC}(\{eq(a, b) \mid a \in \tilde{\Gamma}(t_1, \phi), b \in \tilde{\Gamma}(t_2, \phi)\}) \quad \forall x_1, x_2, \phi \\ &\Downarrow \\ &\tilde{\Gamma}_*(\tilde{e}q_t(t_1, t_2, \phi), \phi) \supseteq \{eq(a, b) \mid a \in \tilde{\Gamma}(t_1, \phi), b \in \tilde{\Gamma}(t_2, \phi)\} \cup \{\perp\} \quad \forall t_1, t_2, \phi \end{aligned}$$

The proof will be a case study of t_1 and t_2 :

1. If $t_1 = all$ or $t_2 = all$ then will $\tilde{e}q_t(t_1, t_2, \phi) = \{all\}$ and the lemma says

$$\tilde{V}_\perp \supseteq \{true, false, \perp\}$$

2. If $t_1 = num$ or $t_2 = num$ we get the relation

$$\{true, false, \perp\} \supseteq \{true, false, \perp\}$$

3. If $t_1 \neq t_2 \wedge atom(t_1) \wedge atom(t_2)$ then will

$$\tilde{e}q_t(t_1, t_2, \phi) = false$$

and

$$\tilde{\Gamma}(t_1, \phi) \cap \tilde{\Gamma}(t_2, \phi) = \{\perp\}$$

hence both sides will be $\{false, \perp\}$.

4. If $t_1 = t_2 \wedge atom(t_1)$ then will

$$\tilde{\Gamma}(t_1, \phi) = \tilde{\Gamma}(t_2, \phi) = \{t_1, \perp\}$$

and both sides will be $\{true, \perp\}$.

5. In other situations will either t_1 or t_2 be a cons expression and

$$\tilde{eq}_t(t_1, t_2, \phi) = \emptyset$$

and either $\tilde{\Gamma}(t_1, \phi)$ or $\tilde{\Gamma}(t_2, \phi)$ would be a set without any atoms. Hence both sides will be $\{\perp\}$

Lemma 5.11. The function \tilde{cond}_s is the term version of \tilde{cond}_* where

$$\begin{aligned} \tilde{cond}_s(e_1, e_2, e_3) = & \text{ (if } true \in e_1 \text{ then } e_2 \text{ else } \emptyset) \cup \\ & \text{ (if } false \in e_1 \text{ then } e_3 \text{ else } \emptyset) \cup \\ & \text{ (if } all \in e_1 \text{ then } lub_s(e_2, e_3) \text{ else } \emptyset) \end{aligned}$$

and

$$\begin{aligned} \tilde{cond}_*(e_1, e_2, e_3) = & \text{ (if } true \in e_1 \text{ then } e_2 \text{ else } \{\perp\}) \cup \\ & \text{ (if } false \in e_1 \text{ then } e_3 \text{ else } \{\perp\}) \cup \\ & \text{ (if } all \in e_1 \text{ then } e_2 \cup e_3 \text{ else } \{\perp\}) \end{aligned}$$

Proof. For $e_1, e_2, e_3 \in \tilde{S}$ we have

$$\tilde{cond}_*(\tilde{\Gamma}_*(e_1, \phi), \tilde{\Gamma}_*(e_2, \phi), \tilde{\Gamma}_*(e_3, \phi))$$

$$\begin{aligned}
&= (\text{if } true \in \tilde{\Gamma}_*(e_1, \phi) \text{ then } \tilde{\Gamma}_*(e_2, \phi) \text{ else } \{\perp\}) \cup \\
&\quad (\text{if } false \in \tilde{\Gamma}_*(e_1, \phi) \text{ then } \tilde{\Gamma}_*(e_3, \phi) \text{ else } \{\perp\}) \cup \\
&\quad (\text{if } all \in \tilde{\Gamma}_*(e_1, \phi) \text{ then } lub_*(\tilde{\Gamma}_*(e_2, \phi), \tilde{\Gamma}_*(e_3, \phi)) \text{ else } \{\perp\}) \\
&= (\text{if } true \in e_1 \text{ then } \tilde{\Gamma}_*(e_2, \phi) \text{ else } \{\perp\}) \cup \\
&\quad (\text{if } false \in e_1 \text{ then } \tilde{\Gamma}_*(e_3, \phi) \text{ else } \{\perp\}) \cup \\
&\quad (\text{if } all \in e_1 \text{ then } lub_*(\tilde{\Gamma}_*(e_2, \phi), \tilde{\Gamma}_*(e_3, \phi)) \text{ else } \{\perp\}) \\
&\subseteq (\text{if } true \in e_1 \text{ then } \tilde{\Gamma}_*(e_2, \phi) \text{ else } \tilde{\Gamma}_*(\emptyset, \phi)) \cup \\
&\quad (\text{if } false \in e_1 \text{ then } \tilde{\Gamma}_*(e_3, \phi) \text{ else } \tilde{\Gamma}_*(\emptyset, \phi)) \cup \\
&\quad (\text{if } all \in e_1 \text{ then } \tilde{\Gamma}_*(lub_s(e_2, e_3, \phi), \phi) \text{ else } \tilde{\Gamma}_*(\emptyset, \phi)) \\
&= \tilde{\Gamma}_*(\tilde{cond}_s(e_1, e_2, e_3), \phi)
\end{aligned}$$

5.4 The Abstract Interpretation.

The \tilde{T} - Interpretation of the language is denoted \mathbf{U}_T . The semantics uses a minimal function graph like method ([Jones & Mycroft, 1986]), but only one value description is found to each function. The semantics is expressed as an interpretation of the framework from section 3.4. The semantics is based on the powerset of terms \tilde{S} defined in section 5.2.

The sets are :

$$\begin{aligned}
D_\sigma &= \tilde{S} && \text{The powerset of terms} \\
D_\tau &= \tilde{S} && \text{the same} \\
\Phi_\sigma &= \tilde{S}^k \times \tilde{S} \\
\Phi_\tau &= \tilde{S}^k \times \tilde{S} \\
C &= \tilde{S}^k && \text{Input Specification} \\
R_\sigma &= D_\sigma \times C^{2n} \\
R_\tau &= D_\tau \times C^{2n}
\end{aligned}$$

The auxiliary functions are

$$\begin{aligned}
atom_\tau(n) &= (\{num'\}, c_0) && \text{if } n \in N \\
&(\emptyset, c_0) && \text{otherwise}
\end{aligned}$$

where $c_0 = \emptyset \times \dots \times \emptyset - 2kn$ times

– This is a raw approximation, but more precise information cannot be used.

$$\begin{aligned}
add_\tau(e_1, \dots, e_k) &(\emptyset, c_0) && \text{if some } e_i = \emptyset \\
&(\{num'\}, c_0) && \text{otherwise}
\end{aligned}$$

$$\text{cond}_\tau((e_1, c_1), (e_2, c_2), (e_3, c_3)) \quad (e_t \cup e_f \cup e_a, c_t \sqcup c_f \sqcup c_a)$$

where

$$\begin{aligned} e_t &= \text{if } true \in e_1 \text{ then } e_2 \text{ else } \emptyset \\ e_f &= \text{if } false \in e_1 \text{ then } e_3 \text{ else } \emptyset \\ e_a &= \text{if } all \in e_1 \text{ then } \{ 'num' \} \text{ else } \emptyset \\ c_t &= \text{if } true \in c_1 \text{ then } c_1 \sqcup c_2 \text{ else } \emptyset \\ c_f &= \text{if } false \in c_1 \text{ then } c_1 \sqcup c_3 \text{ else } \emptyset \\ c_a &= \text{if } all \in c_1 \text{ then } c_1 \sqcup c_2 \sqcup c_3 \text{ else } \emptyset \end{aligned}$$

where \sqcup is elementwise union for cartesian products.

$$\begin{aligned} \text{apply}_{\tau_i}(\theta, (e_1, c_1), \dots, (e_k, c_k)) \quad & (\emptyset, c \text{ if some } e_k = \emptyset \\ & (\theta_i \downarrow 2, c_1 \sqcup \dots \sqcup c_k \sqcup \text{only}_i(e_1, \dots, e_k)) \\ & \text{otherwise} \end{aligned}$$

where $\text{only}_i(v) = (\emptyset_1, \dots, v_i, \dots, \emptyset_n)$

$$\begin{aligned} \text{fetch}_{\sigma_i}(\nu) & \quad (\nu_i, c_0) \\ \text{atom}_\sigma('a) & \quad (a, c_0) \\ \text{apply}_{\sigma_i}(\phi, (e_1, c_1), \dots, (e_k, c_k)) \quad & (\emptyset, c \text{ if some } e_k = \emptyset \\ & (\phi_i \downarrow 2, c_1 \sqcup \dots \sqcup c_k \sqcup \text{only}_{n+i}(e_1, \dots, e_k)) \\ & \text{otherwise} \\ \text{cons}_\sigma(F_1, F_2, (e_1, c_1), (e_2, c_2)) \quad & (\emptyset, c \text{ if } c_2 = \emptyset \vee e_2 = \emptyset \\ & \{ 'cons' \text{ otherwise } \{ F_2 \} \}) \\ \text{basic}_{\sigma_i}(\phi, (e_1, c_1), \dots, (e_k, c_k)) \quad & (\emptyset, c \text{ if some } e_k = \emptyset \\ & (\tilde{op}_{\sigma_i}(e_1, \dots, e_k, \phi), c_1 \sqcup \dots \sqcup c_k) \\ & \text{otherwise} \end{aligned}$$

where \tilde{op}_σ is the term version of the basic operation op_i . The term versions are defined in section 5.3

$$\text{cond}_\sigma((e_1, c_1), (e_2, c_2), (e_3, c_3)) \quad (e_t \cup e_f \cup e_a, c_t \sqcup c_f \sqcup c_a)$$

where

$$\begin{aligned} e_t &= \text{if } true \in e_1 \text{ then } e_2 \text{ else } \emptyset \\ e_f &= \text{if } false \in e_1 \text{ then } e_3 \text{ else } \emptyset \\ e_a &= \text{if } all \in e_1 \text{ then } \text{lub}_s(e_2, e_3, \phi) \text{ else } \emptyset \\ c_t &= \text{if } true \in c_1 \text{ then } c_1 \sqcup c_2 \text{ else } \emptyset \\ c_f &= \text{if } false \in c_1 \text{ then } c_1 \sqcup c_3 \text{ else } \emptyset \\ c_a &= \text{if } all \in c_1 \text{ then } c_1 \sqcup c_2 \sqcup c_3 \text{ else } \emptyset \end{aligned}$$

where lub_s is the term version of lub_* .

$$\text{init}(c, \theta, \phi) \quad (\theta_1 \sqcup (c_1, \emptyset), \dots, (\phi_n \sqcup (c_{2n}, \emptyset)))$$

The iterate function could be defined as

$$\begin{aligned}
& \text{iterate}(\theta, \phi, t_1, \dots, t_k, f_1, \dots, f_k) \\
&= \mathbf{let} ((e_1, d_1), \dots, (e_{2n}, d_{2n})) = \langle t_1(\theta_1 \downarrow 1), \dots, f_n(\phi_n \downarrow 1) \rangle \\
&\quad \mathbf{in} \mathbf{let} c = d_1 \sqcup \dots \sqcup d_{2n} \\
&\quad \mathbf{in} ((c_1, e_1), \dots, (c_{2n}, e_{2n}))
\end{aligned}$$

where both down arrow and index is used to select fields in cartesian products.

Because the two results of \mathbf{E} can be computed independently a definition with the same fixpoint will be

$$\begin{aligned}
& \text{iterate}(\theta, \phi, t_1, \dots, t_k, f_1, \dots, f_k) \\
&= \mathbf{let} c = t_1(\theta_1 \downarrow 1) \downarrow 2 \sqcup \dots \sqcup f_n(\phi_n \downarrow 1) \downarrow 2 \\
&\quad \mathbf{in} ((c_1, t_1(\theta_1 \downarrow 1) \downarrow 1), \dots, (c_{2n}, f_n(\phi_n \downarrow 1) \downarrow 1))
\end{aligned}$$

The advantage of this definition is, that ϕ and τ have a more intuitive meaning: the second field is the result of abstract interpretation with the first field as argument. The second definition is used in the correctness proof.

5.5 Correctness.

The data flow analysis is safe in the following sense:

Let $\mathbf{c} \in \tilde{S}^{k \cdot 2n}$ and assume $S_i \subseteq \tilde{\Gamma}_*(\mathbf{c} \downarrow i \downarrow j, -)$. The input specification \mathbf{c} must not contain cons expressions, what means that the second (unspecified) argument to $\tilde{\Gamma}_*$ is not used.

Let further $\varphi = \mathbf{U}_T \llbracket pgm \rrbracket \mathbf{c}$ ($\varphi = (\theta, \phi)$) then

$$\tilde{\Gamma}_*(\phi_i \downarrow 2, \varphi) \supseteq \tilde{\mathbf{U}}_C \llbracket pgm \rrbracket \downarrow i (S_1, \dots, S_k)$$

Informally this means that if \mathbf{c} is a safe description of the input sets S_1, \dots, S_k then \mathbf{U}_T will give a safe description of the function value found by $\tilde{\mathbf{U}}_C$.

The safeness condition can be restated by use of the following definition

Definition 5.12. A function Γ is defined by

$$\begin{aligned}
\Gamma &: (\tilde{S}^k \times \tilde{S})^{2n} \rightarrow (\wp(\tilde{V}_\perp)^k \rightarrow \wp(N_\perp^\infty))^n \times (\wp(\tilde{V}_\perp)^k \rightarrow \wp(\tilde{V}_\perp))^n \\
\Gamma(\varphi) &= \langle \beta_1, \dots, \beta_n, \Gamma_1(\phi_1, \varphi), \dots, \Gamma_1(\phi_n, \varphi) \rangle \\
\Gamma_1((X_1, \dots, X_k, Y), \varphi) &= \\
&\quad \lambda S_1, \dots, S_k. \mathbf{if} (\forall i : S_i \subseteq \tilde{\Gamma}_*(X_i, \varphi)) \mathbf{then} \tilde{\Gamma}_*(Y, \varphi) \mathbf{else} \tilde{V}_\perp \\
&\quad \beta_i : \lambda S_1, \text{til}, S_k. N_\perp^\infty
\end{aligned}$$

Values of functions called with arguments outside the expected range are described by the value \tilde{V}_\perp as this means, that the value can be anything - we are not able to restrict the set of possible values.

With this function we can formulate the main theorem :

Main Theorem 5.13. For any program and for any input specification \mathbf{c} it is true that

$$\Gamma(\mathbf{U}_T[[pgm]]\mathbf{c}) \downarrow i \sqsupseteq \tilde{\mathbf{U}}_C[[pgm]] \downarrow i \quad i = 1, \dots, 2n$$

where \sqsupseteq is pointwise ordering of functions.

First we notice that the theorem is immediately satisfied for $i = 1, \dots, n$, so we only need to prove the theorem for sort σ . For the proof we define

$$F : (\tilde{S}^k \times \tilde{S})^{2n} \rightarrow (\tilde{S}^k \times \tilde{S})^{2n}$$

$$F(\varphi) = \text{iterate}_T(\text{init}_T(\mathbf{c}, \varphi), \mathbf{E}_{\tau T}[[E_{t1}]]\text{init}_T(\mathbf{c}, \varphi), \dots, \mathbf{E}_{\sigma T}[[E_{fn}]]\text{init}_T(\mathbf{c}, \varphi))$$

and

$$G : (\wp(\tilde{V}_\perp)^k \rightarrow \wp(N_\perp^\infty))^n \times (\wp(\tilde{V}_\perp)^k \rightarrow \wp(\tilde{V}_\perp))^n \rightarrow$$

$$(\wp(\tilde{V}_\perp)^k \rightarrow \wp(N_\perp^\infty))^n \times (\wp(\tilde{V}_\perp)^k \rightarrow \wp(\tilde{V}_\perp))^n$$

$$G(\beta) = \langle \mathbf{E}_{\tau C}[[E_{t1}]]\beta, \dots, \mathbf{E}_{\sigma C}[[E_{fn}]]\beta \rangle$$

Notice that

$$\mathbf{U}_T[[pgm]]\mathbf{c} = LFP(F)$$

and

$$\tilde{\mathbf{U}}_C[[pgm]] = LFP(G)$$

where LFP is the least fixed point operator.

The proof consist of three parts : We want to prove that

$$\forall \varphi : G(\Gamma(\varphi)) \sqsubseteq \Gamma(F(\varphi))$$

The proof requires the lemma

$$\mathbf{E}_{\sigma C}[[E_i]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi) \subseteq \tilde{\Gamma}_*(\mathbf{E}_{\sigma T}[[E_i]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow 1, \varphi)$$

where

$$\tilde{\Gamma}_k : \tilde{S}^k \times (\tilde{S}^k \times \tilde{S})^{2n} \rightarrow \wp(\tilde{V}_\perp)^k$$

$$\tilde{\Gamma}_k(x_1, \dots, x_k, \varphi) = \langle \tilde{\Gamma}_*(x_1, \varphi), \dots, \tilde{\Gamma}_*(x_k, \varphi) \rangle$$

And the theorem is then secured by a fixed point theorem stating that

$$LFP(G) \sqsubseteq \Gamma(LFP(F))$$

The three lemmas will be given here in a rather arbitrary order.

Theorem 5.14. Let A be a set where $F : A \rightarrow A$ has a least fixpoint and B is a complete lattices with a monotonic mapping $G : B \rightarrow B$, With $\gamma : A \rightarrow B$ we assume $G(\gamma(\phi)) \sqsubseteq \gamma(F(\phi)) \quad \forall \phi \in A$ then will $LFP(G) \sqsubseteq \gamma(LFP(F))$

$$\begin{array}{ccc}
 A & \xrightarrow{F} & A \\
 \gamma \downarrow & & \downarrow \gamma \\
 B & \xrightarrow{G} & B
 \end{array}$$

Proof. The proof is strongly related to the proof of Tarski's fixed point theorem [Tarski, 1955] The least fixpoint $LFP(G)$ of a monotonic function G is

$$LFP(G) = \sqcap \{x \in B \mid G(x) = x\}$$

We know that with $\psi = \gamma(LFP(F))$ that $G(\psi) \sqsubseteq \psi$. Let

$$S = \{x \mid x \sqsupseteq G(x)\} \subseteq B$$

then $\psi \in S$. The set of fixpoints of G (call it P) will be a subset of S ($P \subseteq S$). Let $u = \sqcap S$ be the greatest lower bound of the set S in the lattice B and let x be any element in S :

$$x \sqsupseteq u \wedge x \sqsupseteq G(x) \Rightarrow x \sqsupseteq G(u)$$

$G(u)$ is a lower bound of S and hence $u \sqsupseteq G(u)$ and $G(u) \sqsupseteq G(G(u))$, what shows that $G(u) \in S$ and it is greater than $\sqcap S : G(u) \sqsupseteq u$. We conclude that u is a fixpoint of G and it is the least fixpoint.

$$LFP(G) = \sqcap S \sqsubseteq \psi = \gamma(LFP(F)). \quad \square$$

Lemma 5.15. For all function environments φ , and for all input specifications \mathbf{c} , it is true that

$$\begin{aligned}
 & \mathbf{E}_{\sigma C} \llbracket \langle exp \rangle \rrbracket \Gamma(\varphi) \tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi) \\
 & \subseteq \tilde{\Gamma}_*(\mathbf{E}_{\sigma T} \llbracket \langle exp \rangle \rrbracket \varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow 1, \varphi) \\
 & i = 1, \dots, n
 \end{aligned}$$

Proof. The proof will be by structural induction after the expressions $\langle exp \rangle$.

Induction start:

fetch :

$$\begin{aligned} & \mathbf{E}_{\sigma C}[[X_j]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi) \\ &= \tilde{\Gamma}_*((\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow j, \varphi) \\ &= \tilde{\Gamma}_*(\mathbf{E}_{\sigma T}[[X_j]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow 1, \varphi) \end{aligned}$$

atom :

$$\begin{aligned} & \mathbf{E}_{\sigma C}[[a]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi) \\ &= \{a\} \cup \{\perp\} \\ &= \tilde{\Gamma}_*({a}, \varphi) \\ &= \tilde{\Gamma}_*(\mathbf{E}_{\sigma T}[[a]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow 1, \varphi) \end{aligned}$$

Induction step. It is assumed, that the lemma has been proven for all subexpressions.

Basic operations :

$$\begin{aligned} & \mathbf{E}_{\sigma C}[[op(E_1, \dots, E_k)]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi) \\ &= \tilde{op}_*(\mathbf{E}_{\sigma C}[[E_1]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi), \dots, \mathbf{E}_{\sigma C}[[E_k]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi)) \\ &\subseteq \tilde{op}_*(\tilde{\Gamma}_*(\mathbf{E}_{\sigma T}[[E_1]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i), \varphi), \dots, \tilde{\Gamma}_*(\mathbf{E}_{\sigma T}[[E_k]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i), \varphi)) \\ &\subseteq \tilde{\Gamma}_*(\tilde{op}_s(\mathbf{E}_{\sigma T}[[E_1]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow 1, \dots, \mathbf{E}_{\sigma T}[[E_k]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow 1, \varphi), \varphi) \\ &= \tilde{\Gamma}_*(\mathbf{E}_{\sigma T}[[op(E_1, \dots, E_k)]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow 1, \varphi) \end{aligned}$$

Conditionals :

– as for basic operations.

Function calls:

$$\begin{aligned} & \mathbf{E}_{\sigma C}[[\text{call } F_j(E_1, \dots, E_k)]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi) \\ &= \Gamma(\varphi) \downarrow j(\mathbf{E}_{\sigma C}[[E_1]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi), \dots, \\ & \quad \mathbf{E}_{\sigma C}[[E_k]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi)) \\ &= \tilde{\Gamma}_*(\varphi_j \downarrow 2, \varphi) \\ &= \tilde{\Gamma}_*(\mathbf{E}_{\sigma T}[[\text{call } F_j(E_1, \dots, E_k)]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow 1, \varphi) \end{aligned}$$

Lemma 5.16. For all programs and input specifications it is true that

$$\forall \varphi : G(\Gamma(\varphi)) \sqsubseteq \Gamma(F(\varphi))$$

Proof. Three abbreviations can make the proof somewhat simpler:

$$t_i = \mathbf{E}_{\tau T}[[E_{t_i}]]\mathit{init}(\mathbf{c}, \varphi)$$

$$f_i = \mathbf{E}_{\sigma T}[[E_{f_i}]]\mathit{init}(\mathbf{c}, \varphi)$$

$$g_i = \mathbf{E}_{\sigma T}[[E_{f_i}]]\Gamma(\varphi)$$

$$i = 1, \dots, n$$

With this the definition of F can be restated as

$$\begin{aligned} F(\varphi)_{n+i} \downarrow 1 &= ((\mathbf{E}_{\tau T}[[E_{t_1}]]\varphi(\varphi_1 \downarrow 1 \sqcup c_1)) \downarrow 2 \sqcup \dots \sqcup \\ &\quad (\mathbf{E}_{\sigma T}[[E_{f_n}]]\varphi(\varphi_{2n} \downarrow 1 \sqcup c_{2n})) \downarrow 2) \downarrow n + i \\ &= (1_1(\varphi_1 \downarrow 1 \sqcup c_1) \downarrow 2 \sqcup \dots \sqcup f_n(\varphi_{2n} \downarrow 1 \sqcup c_{2n}) \downarrow 2) \downarrow ii \end{aligned}$$

and

$$F(\varphi)_{n+i} \downarrow 2 = f_i(F(\varphi)_{n+i} \downarrow 1) \downarrow 1$$

We want to prove that

$$\forall \varphi : G(\Gamma(\varphi)) \sqsubseteq \Gamma(F(\varphi))$$

or equivalently that for all i

$$g_i \sqsubseteq \Gamma_1(F(\varphi)_{n+i}, F(\varphi))$$

If we assume that

$$S_j \subseteq \tilde{\Gamma}_*(F(\varphi)_i \downarrow 1 \downarrow j, F(\varphi))$$

then we shall prove

$$g_i(S_1, \dots, S_k) \subseteq \tilde{\Gamma}_*(f_i(F(\varphi)_{n+i} \downarrow 1) \downarrow 1, F(\varphi))$$

and due to the monotonicity of g_i we only need to prove

$$g_i(\tilde{\Gamma}_k(F(\varphi)_i \downarrow 1, F(\varphi))) \subseteq \tilde{\Gamma}_*(f_i(F(\varphi)_{n+i} \downarrow 1) \downarrow 1, F(\varphi))$$

or more generally

$$g_i(\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1, \varphi)) \subseteq \tilde{\Gamma}_*(f_i(\varphi_{n+i} \downarrow 1) \downarrow 1, \varphi)$$

Removing the abbreviations this means for $i = 1, \dots, n$:

$$\mathbf{E}_{\sigma T}[[E_{f_i}]]\Gamma(\varphi)\tilde{\Gamma}_k(\varphi_{n+i} \downarrow 1 \sqcup c_i, \varphi) \subseteq \tilde{\Gamma}_*(\mathbf{E}_{\sigma T}[[E_i]]\varphi(\varphi_{n+i} \downarrow 1 \sqcup c_i) \downarrow 1, \varphi)$$

This was shown in the lemma above.

5.6 Compilation.

In section 4.8 we described how to compile a program from the \tilde{V}_1 semantics to the standard semantics. With the information gained from the data flow analysis the compilation can give more effective programs. The straightforward compilation in section 4.8 contained many tests on whether the arguments to `if` expressions and basic operations were the value `'all`. The data flow analysis can find the expressions, that might take the value `all`, and if cannot the test for this value can be removed. The compilation scheme is exemplified below.

Example

The `union` program in the restricted syntax has the following `tbl` program in the restricted syntax.

```

tb (x,y)      = (call t-union (call length-1 x) (call length-1 y))

length-1 (x)  = (if (eq x 0) then nil
                  else (cons (call length-11 x) (call length-12 x)))

length-11(x) = 'all

length-12(x) = (call length-1 (sub x 1))

t-union(x,y) = (if (eq x nil) then 4
                  else
                    (if (call member (car x) y)
                        then (add 12
                              (add (call t-member (car x) y)
                                    (call t-union (cdr x) y))
                              )
                        else (add 15
                              (add (call t-member (car x) y)
                                    (call t-union (cdr x) y))
                              )
                    ))

t-member(x y)= (if (eq y nil) then 4
                  else
                    (if (eq x (car y)) then 9
                        else (add 12 (call t-member x (cdr y))))))

member(x,y)  = (if (eq y nil) then false
                  else
                    (if (eq x (car y)) then true
                        else (call member x (cdr y))
                    ))

```

With the input specification ($\{num'\}, \{num'\}$) to the first function, the data flow analysis will give

tb :

$$(\{num\}, \{num\}) \rightarrow \{num, 4\}$$

length-1:

$$(\{num\}) \rightarrow \{nil, cons(\{length-11\}\{length-12\})\}$$

length-11 :

$$(\{num\}) \rightarrow \{all\}$$

length-12 :

$$(\{num\}) \rightarrow \{nil, cons(\{length-11\}\{length-12\})\}$$

t-union:

$$(\{nil, cons(\{length-11\}\{length-12\})\},$$

t-member:

$$(\{all\}, \{nil, cons(\{length-11\}\{length-12\})\}) \rightarrow \{num, 4\}$$

member

$$(\{all\}, \{nil, cons(\{length-11\}\{length-12\})\}) \rightarrow \{false, all\}$$

With this function environment the subexpressions can be analyzed, and we find

The second condition in **t-union** has the value $\{false, all\}$

The second condition in **t-member** has the value $\{all\}$

The second condition in **member** has the value $\{all\}$

All **car** operations in the program return the value *all*. When we compile the program from the \tilde{V}_\perp - semantics to the standard we use this information. All expressions evaluating to a singleton set are compiled to this value, and conditionals are specialized after how many of the three possible values they can return.

The program can now be compiled to

```
tb (x,y)      = (call t-union (call length-1 x) (call length-1 y))
```

```
length-1 (x) = (if (eq x 0) then nil
                  else (cons 'all (call length-1 (sub x 1))))
```

```
t-union(x,y) = (if (eq x nil) then 4
                  else (add 15
```

```

                                (add (call t-member (car x) y)
                                (call t-union (cdr x) y)
                                )
                                )
                                )

```

```

t-member(x y)= (if (eq y nil) then 4
                  else (add 12 (call t-member x (cdr y))))

```

This is a safe time bound function for the `union` function, but it can be simplified considerably to

```

tb (x,y) = (add 4 (add (mul 18 x) (mul 12 (mul x y))))

```

by removing internal lists and solving finite difference equations. In the rest of the work we are dealing with programs in the standard semantics, what means that traditional program improvement methods can be used.

5.7 Status so far.

Given a program `p` and a size measure *length*. We can construct a program `tbl` such that

$$\tilde{\mathbf{L}} \text{tbl} \langle n_1, \dots, n_k \rangle \geq \max \{ \mathbf{L} tp \langle x_1, \dots, x_k \rangle \mid \forall x_i : \text{length}(x_i) = n_i \}$$

The program `tbl` can be compiled to the `tbp` in the standard semantics

$$\mathbf{L} \text{tb}_p \langle n_1, \dots, n_k \rangle \succeq \max \{ \mathbf{L} tp \langle x_1, \dots, x_k \rangle \mid \forall x_i : \text{length}(x_i) = n_i \}$$

We compile it by making a data flow analysis of the program `tbl`, and examine which expressions might evaluate to the value *'all'*. only such expressions require a special compilation with tests for the value *all*.

The time bound program developed in this way will typically still be in the form of a composition of a step counting version and an inverted length function. The program can be optimized by making a better composition than the structural. A transformation system to make these improvements will be developed in the next chapter.

The time bound program constructed by the methods in chapter 4 and 5 is guaranteed to bound the time requirements safely. The optimizations to be performed concern only the efficiency of the time bound program, but the transformations will not be proven correct. We have already got a safe time bound program, the rest is just sugar.

Ideas and improvements.

The data flow analysis can be extended in at least four ways. The extensions might be interesting if the analysis were used in other areas where we want an approximation of a collecting semantics for a non-flat domain.

1. The analysis requires that all constants are atomic to secure termination. We can allow composite constants if the set of terms is extended with these constants and *all their substructures*. This will still give a finite set of terms. The 'car' and 'cdr' operation on such constants should then return the substructure, but the 'cons' operation should not be changed.

2. A variable will always be described by a finite set of terms in the variable environment. When we are analyzing the branches of a conditional we can find out which terms in the environment will make the predicate *true* or *all* and only use them in the first branch. The environment can be restricted in the same way for the second branch.

3. The underlying domains in the analysis can be changed. We can approximate the analysis to a Known/Unknown analysis (see [Jones & Mycroft, 1986] and [Sestoft, 1986]), by using a simplified termset

$$\tilde{T} = \{all\} \cup \{K\} \cup \{cons\} \times \mathbf{P}(Fn) \times \mathbf{P}(Fn)$$

where *all* plays the role of *Unknown* (meaning possibly unknown) and *K* denotes a *Known* value. A *cons* expression denotes a non-atomic value whose K/U property is described by the function set of the arguments.

The $\tilde{\Gamma}$ function is changed to

$$\begin{aligned} \tilde{\Gamma}(all, \phi) &= \tilde{V}_\perp \\ \tilde{\Gamma}(K, \phi) &= \tilde{V} \\ \tilde{\Gamma}(cons(g_1, g_2)) &= \text{— as before} \end{aligned}$$

This means that *all* in $\mathbf{P}\mathbf{P}(V)$ denotes $\mathbf{P}(V)$ and *K* denotes $\{\{x\} | x \in V\}$ (see figure 5.1).

4. As the set \tilde{S} is finite we can extend the function environment to $\Phi = (\tilde{S} \rightarrow \tilde{S})^n$. Also this function environment will be finite, and it will give more detailed descriptions, distinguishing between different applications of functions. It will be interesting to see what type of information one can get from this type of analysis.

Chapter 6

A Concept of Driving

Last chapter resulted in a time bound program that in the standard semantics computes a safe time bound function. Hence we have now reduced the problem of generating closed form time bound functions to a program transformation problem. This chapter shows a simple way to improve the time bound program. Other more powerful program transformation systems should be able to give supplementary optimizations. The implemented system has the advantage of being fully automatic (no annotation) and is able to produce closed form expressions for a number of simple programs. A central part of the optimizations is to solve finite-difference equations. This part is described in the next chapter as it requires a more comprehensive treatment.

As the time bound programs at this step typically will consist of the step counting version syntactically composed with inverted length functions (after the translation to the standard semantics), a transformation system should improve the program by a better composition. Another way to say it is that the step counting version should be specialized with arguments computed by the inverted length functions.

This is in contrast to the MIX project (see [Sestoft, 1986]) where the functions are specialized when some arguments are known. This is also true at the external level: It is not $\mathbf{L} \text{mix} \langle \mathbf{t}_p, \text{length-1} \rangle$ we are interested in but rather something like

$$\lambda x. \mathbf{L} \text{mix} \langle \mathbf{t}_p, LL \text{length-1} \langle x \rangle \rangle$$

And to this the MIX system in its present state is of no use.

The idea of specializing functions with functions as arguments can be found as procedure-expressions in [Scherlis, 1981] and in super-compilation and driving [Turchin, 1986]. Also [Wegbreit, 1976] describes a way to improve function composition by matching subgoals in unfolded expressions from more general goals. The works by Scherlis and Wegbreit describes transformation rules, but they cannot answer the question about in which order the rules should be applied. Supercompilation gives a general answer to this question although the set of basic configurations are difficult to characterize. Another problem with driving is that it is designed for the language REFAL, and it cannot easily be generalized to other languages. In this work we use a variant of driving to be used for the present language

and the basic configurations are function composition similar to the methods used by Scherlis and Wegbreit. The transformation system is described from section 6.2 but first let us give an example.

6.1 6.1 Example

The example here is taken from [Wegbreit 1976]. Given the following definition of `append` :

```
append(x,y) = (if (eq x nil) then y
                 else (cons (car x) (call union (cdr x) y)) )
```

Consider appending three lists by the function

```
f(x,y,z) = (call append (call append x y) z)
```

We try to improve this little program by analyzing the right hand side of the first function:

```
(call append (call append x y) z)
```

Here we immediately get the first function composition, and this is viewed as a basic configuration (or goal or procedure-expression). By functions we here mean the user defined functions, everything else is considered as basic expressions, as they cannot be unfolded in any way. The configuration is represented as `[append(append, var)]` where `var` denotes an arbitrary expression. The configuration is now viewed as a function and analyzed without its original context. This is in contrast to [Turchin, 1986] where the configurations first are recognized as being basic at the second appearance, hence the first reduction is done in the original context. The present method is chosen because of its simpler termination properties.

The configuration contains three free variables, and as in [Turchin, 1986] we drive the free variables forcefully through the expressions of the program, until the expression represents a *progressive* method for computing the configuration it defines (cp. [Scherlis, 1981]):

```
[append(append, var)](a, b, c) =
```

```
(if (null a) then (call append b c)
    else (cons (car a) (call append (call append (cdr a) b) c))
```

The driving is done by constructing a conditional expression with the first condition to be performed in a normal order evaluation of the expression. This method resembles the idea of outside-in reductions in driving REFAL [Turchin, 1986].

This expression contains two configurations: The dummy one:

```
[append(var, var)]
```

and the original configuration:

```
[append(append, var)]
```

Hence we loop back and get the new function definition:

```
append3(a,b,c) = (if (null a) then (call append b c)
                  else (cons (car a) (call append3 (cdr a) b c)))
```

Inserting in the original context the program is improved to

```
f(x,y,z)      = (call append3 x y z)
```

```
append(x,y)   = (if (eq x nil) then y
                  else (cons (car x) (call union (cdr x) y)) )
```

```
append3(a,b,c) = (if (null a) then (call append b c)
                  else (cons (car a) (call append3 (cdr a) b c)))
```

6.2 Basic Configurations

When unfolding function calls in program transformation systems certain criteria are needed to secure termination. The most restrictive rule (after no unfolding) would be only to allow one unfolding per function in each branch. Here we use a somewhat extended principle by only allowing one unfolding per *basic configuration* in each branch. By only defining a finite number of basic configurations termination is easily secured. In this section we define the basic configurations and show how progressive computation schemes can be established. In the next section the driving algorithm is described.

The set of basic configurations is defined as equivalence classes in the set of expressions, and two expressions are equivalent if they are function calls with call to the same functions as arguments. The configuration can be characterized by the first function name and a list of arguments with either a function name or an indication that the argument can be anything. The set of basic configurations can be extended with constants as arguments, but to secure termination there are some limitations: once a constant is used as argument only subexpressions of it may appear in basic configurations in a branch. We return to this in the next section.

The basic configuration can be represented by its most general expression: the expression with free variables for all unspecified subexpressions. In this section we want to construct a new expression to compute the value of the expression. If the program was well constructed from the start the first definition is not expected

to be improved. The new expression should give a progressive ([Scherlis, 1981]) definition of the basic configuration, what means that it must not be defined by itself. An easy way to obtain this would be to unfold all the functions in the expression once, but as the function compositions are important in this method unfolding should be done with care. There is a better method to construct the expression:

1. Search the expression in normal order (outside-in) for the first basic condition to be evaluated. A condition is basic if it does not contain any calls to user defined functions.
2. In the context of the condition respectively true and false the arguments to the first function call is reduced (and unfolded) as much as possible without getting any conditional expressions.
3. If the two new argument sets are changed then insert them in the function call getting two expression. They are reduced as much as possible without getting any conditional expressions, and *without removing any function compositions*. A conditional expression is now constructed by the condition found above and these two expressions. The last requirement is stronger than normally in driving, but it is due to our restricted set of basic configurations. The first requirement is equivalent to reduction of *transient* configurations in supercompilation.
4. If the two argument sets could not use the information from the condition then unfold the first function and insert the arguments. The expression is then reduced as much as possible as in (3).

Reductions.

“Reduce as much as possible” means the application of a number of standard reduction schemes (the names are taken from [Wegbreit, 1976]):

Local simplification rules. This is also called symbolic evaluation (in [Sestoft, 1986]). For example can a *car* operation with a *cons* expression as argument be reduced to the first argument of the *cons* expression.

Distributing conditionals. This is also called reduction to normal form (in [Wegbreit, 1975]). The conditionals are taken out of arguments to basic operations and functions, so they appear first in expressions. This makes the next step easier.

Evaluating in context. In the simplest form this consist in removing duplicate conditions in expression. When a function call is expanded we know that all the predicates in conditionals leading to the expression true.

Expanding function bodies. This is also called application (in [Scherlis, 1981]) or unfolding. The method must be used with care, and we adopt a criteria similar to [Wegbreit, 1976]: Functions are only expanded if they give condition free expressions and if they do not remove function compositions in the program. There are still a possibility for nontermination: if the program contains branches with nonterminating evaluations, and these branches are not used because of unanalyzable but constant predicates, then these expression might be tried evaluated by the reduction algorithm. However, this fact has not been a problem for the present application of the transformation system.

Example.

In the example with the `append` function the expression for the configuration `[append(append, var)]` was

```
(call append (call append a b) c)
```

The first basic condition to be performed in a normal order evaluation is `(null a)`. Assume the condition is true, then will the argument set `((call append a b) c)` reduce to `(b c)`, giving the call

```
(call append b c)
```

If the condition is false the arguments reduces to

```
( (cons (car a) (call append (cdr a) b)) c )
```

and the call

```
(call append (cons (car a) (call append (cdr a) b)) c)
```

can be reduced to

```
(cons (car a) (call append (call append (cdr a) b) c))
```

Finally we have the expression

```
(if (null a) then (call append b c)
    else (cons (car a) (call append (call append (cdr a) b) c)))
```

6.3 Driving algorithm

The driving algorithm starts by the right hand side of the first function in the program. All basic configurations are examined from outside in, and progressive definition of the configurations (found in the last section) are analyzed in the same way recursively. To secure termination of the driving algorithm a basic configuration is only analyzed once in each branch, and to improve the residual program, three operations are performed: generalization of basic configurations, recognition

of earlier defined function and solving difference equations. The last type of reductions are required because of our special class of programs with numbers as domain and range. The algorithm is described in two parts: a local one dealing with the analysis of a single basic configuration, and the global one controlling generalization.

Local part. The driving algorithm treats basic configuration in the following five steps.

1. Given a basic configuration.
2. Construct a progressive computation scheme for it (see previous section).
3. Locate and drive basic configurations in subexpressions by the global part of the driving algorithm.
4. Locate call to the original basic configuration. If there are such recursive calls then the expression is a recursive definition of the basic configuration viewed as a function in its free variables. This definition is then examined by:
 5. (a) Matching it with function definitions in the original program. If it is defined in advance then substitute the expression with a call to this function.
 - (b) Solve difference equations by matching the definition with the rules given in the next chapter.
 - (c) Otherwise construct a new recursive function and substitute the expression with a call to this function. (Folding).
6. If no recursive calls in the expression then do nothing.

Generalization. In the implemented system only a simple sort of generalization is used: If a basic configuration contains a constant as argument and the algorithm when examining the expression reaches a basic configuration with another constant as argument, then the first configuration is generalized withvar as an argument denoting an arbitrary expression.

The method can be extended to generalize function compositions by replacing functions as arguments withvar if this makes it possible to fold basic configurations to new functions (see point 4.3 above).

Global part. The global part of the driving algorithm controls the program transformation system.

1. Start with the right hand side of the first function in the program.
2. Search the expression from outside in for basic configuration.

3. If the configuration is a generalization of an identified, but suspended configuration, then backtrack and generalize the first identified (outermost) configuration. If the configuration is either dummy (no function compositions) or identical with a suspended configuration, then loop back by returning the expression.
4. Otherwise analyze the configuration with the local part of the driving algorithm.
5. If the resulting expression is a conditional expression then the basic configuration was no success and continue driving with the arguments of the basic configuration (the original call).
6. Otherwise reduce the expression in its context and continue driving with this expression.

6.4 Status

The present program transformation system is concise and deterministic. For some time bound programs the system will produce closed-form expressions (non-recursive definitions), Some would be improved by removing intermediate lists produced by `length-1` functions and consumed by the `tp` function, and for the rest no optimization (and no aggravation) is expected. The transformation system is not tried proven correct and with the present description it would not be easy to do that. The missing safeness proof is not a major drawback of the work since we already in the last chapter obtained a safe time bound program.

Example.

Let us show the idea of driving by this non trivial example, where it is more complicated to remove the intermediate lists. The algorithm described in earlier chapters will produce this time bound function for the program `reverse`:

```
time (x) = (call t-reverse (call length-1 x))

length-1 (x) =
  (if (eq x 0) then nil
      else (cons 'all (call length-1 (sub x 1))))

reverse (x) =
  (if (null x) then nil
      else (call append (call reverse (cdr x)) '(all)))

t-append (x) =
  (if (null x) then 4
      else (add 10 (call t-append (cdr x))))
```

```

append (x y) =
  (if (null x) then y
      else (cons 'all (call append (cdr x) y)))

t-reverse (x) =
  (if (null x) then 4
      else (add 11
                (add (call t-reverse (cdr x))
                     (call t-append (call reverse (cdr x))) )))

```

To a start we try to solve difference equations in the program. It is convenient to do this before using the driving algorithm because we else might solve the same difference equation many times in the algorithm. There is only one function in the program to be simplified by the rules presented in the next chapter:

```

t-append (x) =
  (add 4 (mul 10 (call length x)))

```

where `length` is the usual length function from Lisp.

The first basic configuration is `[t-reverse(length-1)]` with the expression

```

[t-reverse(length-1)] (x) =
  (if (eq x 0) then 4
      else (add 15
                (add (call t-reverse (call length-1 (sub x 1)))
                     (mul 10 (call length (call reverse (call length-1 (sub x 1))))))
          )))

```

The call to `t-reverse` can be folded back, and we proceed with the configuration

```

[length(reverse)] (x) =
  (if (null x) then 0
      else (call length (call append (call reverse (cdr x)) '(all))))

```

giving the new configuration

```

[length(append)] (x,y) =
  (if (null x) then (call length y)
      else (add 1 (call length (call append (cdr x) y))))

```

This difference equation can be solved giving

```

[length(append)] (x,y) =
  (add (call length x) (call length y))

```

Inserted in the context from the previous configuration we get

```

[length(reverse)] (x) =
  (if (null x) then 0
      else (add 1 (call length (call reverse (cdr x)))))

```

giving

```
[length(reverse)] (x) = (call length x)
```

Inserted in the original context we get

```
[t-reverse(length-1)] (x) =
  (if (eq x 0) then 4
      else (add 15
              (add (call t-reverse (call length-1 (sub x 1)))
                    (mul 10 (call length (call length-1 (sub x 1)))
                      )))
  )))
```

Where we get the new configuration

```
[length(length-1)] (x) =
  (if (eq x 0) then 0
      else (add 1 (call length (call length-1 (sub x 1)))))
```

easily solved to

```
[length(length-1)] (x) = x
```

And the original configuration is now

```
[t-reverse(length-1)] (x) =
  (if (eq x 0) then 4
      else (add 15
              (add (call t-reverse (call length-1 (sub x 1)))
                    (mul 10 (sub x 1))
                      )))
```

or simplified

```
[t-reverse(length-1)] (x) =
  (if (eq x 0) then 4
      else (add 5 (add (mul 10 x)
                      (call t-reverse (call length-1 (sub x 1)))
                      )))
```

This difference equation can be solved to

```
[t-reverse(length-1)] (x) =
  (add 4 (add (mul 10 x) (mul 5 (mul x x)) ))
```

Finally the program can be reduced to

```
time (x) = (add 4 (add (mul 10 x) (mul 5 (mul x x)) ))
```

Ideas and improvements.

If the transformation system should be used in other contexts a better generalization principle is needed. In the present applications it has not been needed. Assume as example that under driving a configuration $[f(g, h)]$ the configuration $[f(g, i)]$, and if the driving does not reach other calls to the configuration $[f(g, h)]$ then the original configuration should be generalized to $[f(g, var)]$. This idea can be found in all three references ([Scherlis, 1981], [Turchin, 1986], and [Wegbreit, 1976]).

The transformation system has some similarities with abstract interpretation and word algebra. The basic configurations are extended expressions where the symbol var is used as a top element denoting any expression (as all in \tilde{V}_\perp). Generalization of basic configurations can then be viewed as taking least upper bounds in some sort of an extended word algebra (containing var). In this way the reduction rules (local simplification rules) can be seen as basic operations in the word algebra. It would be interesting to see whether this is just a similarity or the idea can be used to formalize the driving principle.

Chapter 7

Finite-Difference Equations

To get reasonable time bound functions it is important to be able to automatically solve difference equations. In the work by Wegbreit [Wegbreit, 1975] the system contained a collection of transformation rules based on well-known formulae for arithmetic and geometric series. A large system to solve finite difference equations can be found in [Cohen & Katcoff, 1977]. The standard formulae found in any mathematical handbook can be very useful in such transformation packages, but not all rules give optimized computation when they are used in program transformation. As example will

$$\prod_{i=\ell}^n i = \frac{n!}{(\ell - 1)!}$$

with n as a free variable gives a conceptually simpler expression, but computationally it is worse on a machine dealing only with the four basic arithmetic operations. Hence it should not be included in a program transformation system but it might be interesting in another context. The mathematically based transformation rules are often used in a way, where it is not so important whether bounds of series are known constants or variables, but in program transformation this point gets very important.

In this chapter eight transformation rules are developed. They give optimized computation, and they are designed to a language only allowing the four standard arithmetic operations. Other transformation rules could of course be useful if exponentiation and the factorial function were basic functions in the language.

7.1 Classes of finite-difference equations.

A major problem in the construction of time bound functions is to solve finite-difference equations. A finite-difference equation is a recursive function definition, where the function value depends on the argument $n \in N$ and on the function values for smaller values than n

$$F(n) = H(n, F(n - 1), F(n - 2), \dots, F(0))$$

We cannot solve such equations in general, but for certain classes of definitions H there are simple solutions. We will concentrate on two classes of difference equations :

$$F(n) = \mathbf{if } n = \ell \mathbf{ then } c_1 \mathbf{ else } f(n) + g(n) \cdot F(n - 1)$$

$$F(x) = \mathbf{if } \mathit{null}(x) \mathbf{ then } c_1 \mathbf{ else } c_2 + c_3 \cdot F(\mathit{dec}(x))$$

where ℓ, c_1, c_2 and c_3 are constants (or arguments, which don't change in recursive calls), and null , dec , f and g are functions. The second scheme can also be used for non-numeric functions as null and dec need not be numeric functions.

A scheme similar to the first is often seen. We state this as the first transformation rule

Rule 7.1.

$$\begin{aligned} F(n) &= \mathbf{if } n = \ell \mathbf{ then } c_1 \mathbf{ else } f(n) + g(n) \cdot F(n + 1) \\ &= G(n, \ell) \end{aligned}$$

where $G(n, m) = \mathbf{if } n = m \mathbf{ then } c_1 \mathbf{ else } f(m) + g(m) \cdot G(n, m - 1)$

This is a difference equation in m and it is of the first class.

7.2 Simple decrement.

Equations for finite summations and product series can be used as models for solution of finite difference equations. These two rules give the relationship:

$$\begin{aligned} F(x) &= \sum_{i=\ell}^x f(i) \\ &= \mathbf{if } x = \ell - 1 \mathbf{ then } 0 \mathbf{ else } f(x) + F(x - 1) \end{aligned}$$

and

$$\begin{aligned} F(x) &= \prod_{i=\ell}^x f(i) \\ &= \mathbf{if } x = \ell - 1 \mathbf{ then } 1 \mathbf{ else } f(x) \cdot F(x - 1) \end{aligned}$$

An even larger class of difference equations can be solved by the scheme below containing both summation and product series. The formula is known as the *arithmetic-geometric* series. (see [Cohen & Katcoff, 1977])

$$\begin{aligned} F(x) &= \mathbf{if } x = \ell \mathbf{ then } c_1 \mathbf{ else } f(x) + g(x) \cdot F(x - 1) \\ &= c_1 \prod_{i=\ell+1}^x g(i) + \sum_{i=\ell+1}^x \left(f(i) \cdot \prod_{j=\ell+1}^i g(j) \right) \end{aligned}$$

Proof. The relation is shown by induction in the difference between x and ℓ :

For $x = \ell$ we have

$$\begin{aligned} F(\ell) &= c_1 \prod_{i=\ell+1}^x g(i) + \sum_{i=\ell+1}^{\ell} \left(f(i) \cdot \sum_{j=\ell+1}^{\ell} g(j) \right) \\ &= c_1 \cdot 1 + 0 = c_1 \end{aligned}$$

For $x > \ell$ we have under the assumption, that the relation is shown for $x - 1$:

$$\begin{aligned} F(x) &= \mathbf{if } x = \ell \mathbf{ then } c_1 \mathbf{ else } f(x) + g(x) \cdot F(x - 1) \\ &= f(x) + g(x) \cdot \left(c_1 \prod_{i=\ell+1}^{x-1} g(i) + \sum_{i=\ell+1}^{x-1} \left(f(i) \cdot \prod_{j=\ell+1}^{x-1} g(j) \right) \right) \\ &= c_1 \prod_{i=\ell+1}^x g(i) + f(x) + \sum_{i=\ell+1}^{x-1} \left(f(i) \cdot \prod_{j=\ell+1}^x g(j) \right) \\ &= c_1 \prod_{i=\ell+1}^x g(i) + \left(f(i) \cdot \prod_{j=\ell+1}^x g(j) \right) \end{aligned}$$

What was to be proven.

This transformation rule is certainly not an optimization in the general case. The original definition has a time bound of magnitude $x - \ell$, and the transformed expression requires of magnitude $(x - \ell)^2$ operations. Nevertheless there are **some** interesting special cases of this scheme giving optimized computation. In the first relation the function g is assumed to be constant.

$$\begin{aligned} F(x) &= \mathbf{if } x = \ell \mathbf{ then } c_1 \mathbf{ else } f(x) + c_2 F(x - 1) \\ &= c_1 \cdot c_2^{x+\ell} + \sum_{i=\ell+1}^x f(i) \cdot c_2^{x-i} \end{aligned}$$

From this relation four special cases follows:

Rule 7.2.

$$\begin{aligned} F(x) &= \mathbf{if } x = \ell \mathbf{ then } c_1 \mathbf{ else } c_2 + F(x - 1) \\ &= c_1 + c_2(x - \ell) \\ &= c_2 \cdot x + (c_1 - c_2 \cdot \ell) \end{aligned}$$

Rule 7.3.

$$\begin{aligned}
F(x) &= \mathbf{if } x = \ell \mathbf{ then } c_1 \mathbf{ else } c_2 + c_3 \cdot x + F(x - 1) \\
&= c_1 \cdot c_3^{x-\ell} + \sum_{i=\ell+1}^x (c_2 + c_3 \cdot i) \\
&= c_1 + c_2(x - \ell) + c_3(x - \ell)(x + \ell + 1)/2 \\
&= \frac{c_3}{2} \cdot x^2 + \left(c_2 + \frac{c_3}{2}\right) \cdot x + \left(c_1 - \frac{c_3}{2} \cdot \ell(\ell - 1)\right)
\end{aligned}$$

Rule 7.4.

$$\begin{aligned}
F(x) &= \mathbf{if } x = \ell \mathbf{ then } c_1 \mathbf{ else } c_2 + c_3 \cdot F(x - 1) \\
&= c_1 \cdot c_3^{x-\ell} + \sum_{i=\ell+1}^x (c_2 \cdot c_3^{i-\ell}) \\
&= c_3^{x-\ell} \cdot \left(c_1 - \frac{c_2}{1 - c_3}\right) + \frac{c_2}{1 - c_3} \\
&= G(x) \cdot \left(c_1 - \frac{c_2}{1 - c_3}\right) + \frac{c_2}{1 - c_3}
\end{aligned}$$

where $G(x) = \mathbf{if } x = \ell \mathbf{ then } 1 \mathbf{ else } c_3 \cdot G(x - 1)$.

Rule 7.5.

$$\begin{aligned}
F(x) &= \mathbf{if } x = \ell \mathbf{ then } c_1 \mathbf{ else } c_2 + c_3 \cdot x + c_4 \cdot F(x - 1) \\
&= c_1 \cdot c_4^{x-\ell} + \sum_{i=\ell+1}^x (c_2 + c_3 \cdot i) \cdot c_4^{x-i} \\
&= c_4^{x-\ell} \cdot \left(\frac{c_2 + \ell \cdot c_3}{c_4 - 1} + c_1 + \frac{c_3 \cdot c_4}{(c_4 - 1)^2}\right) - \left(\frac{c_2 + x \cdot c_3}{c_4 - 1} + \frac{c_3 \cdot c_4}{(c_4 - 1)^2}\right) \\
&= G(x) \cdot \left(\frac{c_2 + \ell \cdot c_3}{c_4 - 1} + c_1 + \frac{c_3 \cdot c_4}{(c_4 - 1)^2}\right) - x \cdot \frac{c_3}{c_4 - 1} - \left(\frac{c_2}{c_4 - 1} + \frac{c_3 \cdot c_4}{(c_4 - 1)^2}\right)
\end{aligned}$$

where $G(x) = \mathbf{if } x = \ell \mathbf{ then } 1 \mathbf{ else } c_4 \cdot G(x - 1)$

In developing these rules the following formulae have been used ([Spiegel, 1968]):

$$\begin{aligned}
\sum_{i=0}^{n-1} (a + i \cdot d) &= \frac{1}{2} \cdot n \cdot (2a + (n - 1) \cdot d) \\
\sum_{i=0}^{n-1} (a \cdot r^i) &= \frac{a(1 - r^n)}{1 - r} \\
\sum_{i=0}^{n-1} (a + i \cdot d)r^i &= \frac{a(1 - r^n)}{1 - r} + \frac{rd(1 - nr^{n-1} + (n - 1)r^n)}{(1 - r)^2}
\end{aligned}$$

7.3 General decrement.

In the previous section we have seen rules where the function value for argument x depends on the function value for $x - 1$. In the general case the solutions are much more limited, but for certain cases we can make useful relations. There are two variants of rule 2 and 4.

Rule 7.6.

$$\begin{aligned} F(x) &= \mathbf{if\ } null(x) \mathbf{\ then\ } c_1 \mathbf{\ else\ } c_2 + F(dec(x)) \\ &= c_1 + c_2 \cdot G(x) \end{aligned}$$

where $G(x) = \mathbf{if\ } null(x) \mathbf{\ then\ } 0 \mathbf{\ else\ } 1 + G(dec(x))$

Rule 7.7.

$$\begin{aligned} F(x) &= \mathbf{if\ } null(x) \mathbf{\ then\ } c_1 \mathbf{\ else\ } c_2 + c_3 \cdot F(dec(x)) \\ &= G(x) \cdot (c_1 + c_2/(c_3 - 1)) - c_2/(c_3 - 1) \end{aligned}$$

where $G(x) = \mathbf{if\ } null(x) \mathbf{\ then\ } 1 \mathbf{\ else\ } c_3 \cdot G(dec(x))$

The last rule can of course only be used for $c_3 \neq 1$, but for $c_3 = 1$ the first rule can be used.

Logarithmic decrement.

A special equation is given by the scheme

$$F(x) = \mathbf{if\ } x = 1 \mathbf{\ then\ } c_1 \mathbf{\ else\ } f(x) + c_2 \cdot F(x \text{ div } 2)$$

where we have the transformation:

$$F(x) = c_2^{\log(x)} \cdot c_1 + \sum_{i=0}^{\log(x)-1} c_2^i \cdot f(x \text{ div } 2^i)$$

where $\log(x) = \mathbf{if\ } x = 1 \mathbf{\ then\ } 0 \mathbf{\ else\ } 1 + \log(x \text{ div } 2)$. A special case is

Rule 7.8.

$$\begin{aligned} F(x) &= \mathbf{if\ } x = 1 \mathbf{\ then\ } c_1 \mathbf{\ else\ } c_3 + c_2 \cdot F(x \text{ div } 2) \\ &= c_2^{\log(x)} \cdot c_1 + \sum_{i=0}^{\log(x)-1} c_2^i \cdot c_3 \\ &= G(x) \cdot \left(c_1 - \frac{c_3}{1 - c_2} \right) + \frac{c_3}{1 - c_2} \end{aligned}$$

where

$$\begin{aligned} G(x) &= c_2^{\log(x)} \\ &= \mathbf{if\ } x = 1 \mathbf{\ then\ } 1 \mathbf{\ else\ } c_2 \cdot G(x \text{ div } 2) \end{aligned}$$

Other schemes.

This list of transformation rules is of course not complete. Even under the restriction of having a limited set of basic operations and requiring optimized computation there are many other rules. All rules here work with only one argument, but there are also interesting difference equations with two or more variables. Difference equations of higher order (depending on more than one previous element) can in some situation be solved by use of characteristic polynomials (see [Cohen & Katcoff, 1977]).

Chapter 8

Results

The algorithm described in chapter 4 - 7 have been realized in Franz Lisp [Foderaro et al., 1983] on a Vax. The programs are written in an embedded language and translated to Lisp. The source text can be found in appendix C. In appendix A we show the results from analysis of some programs, and in appendix B we present test runs with generated time bound programs. The results are compared with execution times for some arbitrarily chosen inputs of the same size.

8.1 Results.

In this section we present results from experiments with the system. We give examples of generated time bound programs and we test their efficiency for several inputs.

The algorithm is divided in three phases, and after each we can extract a time bound program:

Phase 1. The result of naive compilation of `tbl` to the standard semantics (see section 4.8).

Phase 2. The result of compilation using the data flow analysis developed in chapter 5.

Phase 3. The final time bound program after improvements by driving (chapter 6 and 7).

In the examples below we are comparing the running time of the three different time bound programs.

Lookup.

The first example is the program `lookup`. The program is defined as:

```
lookup (x y) =  
  (if (null y) then nil
```

```

else
  (if (eq (car (car y)) x)
      then (car y)
      else (call lookup x (cdr y))))))

```

The program searches for a binding in an A-list. It is classically known as `assq`. As inverted length functions we specify:

```
all-1 (x) = 'all
```

for the first argument and for the second argument

```
length-1 (x) = (if (eq x 0) then nil
                  else (cons all (call length-1 (sub x 1))))

```

The analyzer produces the following time bound program :

```
time (x y) = (add 4 (mul 13 y))
```

The analysis took 7 seconds machine time.

In the table below we show the results of running the three different time bound programs for `lookup` with second input of length 4. We show also how long time it took for the time bound programs to find the result:

	Phase 1	Phase 2	Phase 3
Result	56	56	56
Time used	448	97	5

All times are measured in number of basic operations. The time bounds should be compared with executions of the original program with input of the same size:

```
(lookup 8 ((1) (2) (8) (10))) => (8)
```

Time used : 37

```
(lookup 8 ((1) (2) (7) (10))) => nil
```

Time used : 56

We can see that the time bound is equivalent to the maximal running time of the program. This is achieved when the first argument does not appear in the list in the second argument.

Union.

The analysis of the `union` program is described in chapter 2. The generated time bound program is

```
time (x,y) = (add 4 (mul (add 19 (mul 12 v)) u))
```

The analysis took 11 seconds.

Timebound of `union` with first input of length 3 and second input of length 4:

	Phase 1	Phase 2	Phase 3
Result	205	205	205
Time used	6971	1006	9

Examples

`(union (3 4 5) (1 2 6 7)) => (3 4 5 1 2 6 7)`

Time used : 205

`(union (3 4 5) (5 3 7 1)) => (5 3 7 1)`

Time used : 125

The time bound is the running time of the program with two disjoint sets as argument.

Reverse.

The program `reverse` is described in chapter 6 of this paper. The resulting time bound program was:

```
time (x) = (add 4 (add (mul 10 x) (mul 5 (mul x x))))
```

The analysis took 18 seconds.

Timebound of `reverse` with input of length 4:

	Phase 1	Phase 2	Phase 3
Result	124	124	124
Time used	1292	322	11

Examples

`(reverse (1 2 6 7)) => (7 6 2 1)`

Time used : 124

`(reverse (7 6 2 1)) => (1 2 6 7)`

Time used : 124

The running time of this program is only depending on the length of the list.

Parse.

The next example is a deterministic parser. From a little grammar we construct an SLR(1) parser table. The parser is programmed in Lisp by folding the rows in the table (see [Aho & Johnson, 1974]).

The grammar is:

- 0 : $e' \rightarrow e$
- 1 : $e \rightarrow -t$
- 2 : $e \rightarrow t$
- 3 : $t \rightarrow t + f$
- 4 : $t \rightarrow f$
- 5 : $f \rightarrow f * s$
- 6 : $f \rightarrow s$
- 7 : $s \rightarrow id$

From this we construct the SLR(1) table:

	-	+	*	id		e	t	f	s
1	s3			s7		2	4	5	6
2					r0				
3				s7			8	5	6
4		s9			r2				
5		r4	s10		r4				
6		r6			r6				
7		r7	r7		r7				
8		s9			r1				
9				s7				11	6
10				s7					12
11		r3	s10		r3				
12		r5	r5		r5				

The parser is written tail-recursively and it builds a prefix representation for the expressions on an attribute stack. The program can be found in appendix A together with the generated time bound program. The time bound program is not

in closed form, as it requires a more extensive transformation package to solve the equation system. The analysis took 38 minutes machine time. Most of the time was spent in the data flow analysis.

Timebound of `parse` with input of length 3:

	Phase 1	Phase 2	Phase 3
Result	482	482	525
Time used	6445	1182	909

Notice the time bound found in phase 3 is higher than in the previous phases. This is because one of the optimizations in the transformation package does not preserve total correctness but only the monotonicity: all identical elements and *constants* are moved out of call to 'max'. This can always be done safely for time bound programs and it will normally only give a little change in the time bound. We give a short description of the transformations in the next section.

Examples

`(parse (6 * 7)) => (* 6 7)`

Time used : 451

`(parse (- 3 + 6 * 7)) => (- (+ 3 (* 6 7)))`

Time used : 840

Bubble sort.

Finally the system has been tried with a naive bubble sort algorithm. The program and the time bound program are shown in appendix A. The analysis took 46 seconds.

Timebound of `.h sort` with input of length 4:

	Phase 1	Phase 2	Phase 3
Result	1773	1773	1773
Time used	60853	11398	517

Examples

`(sort (5 3 7 1)) => (1 3 5 7)`

Time used : 1203

`(sort (7 6 2 1)) => (1 2 6 7)`

Time used : 1527

The time bound program was not in closed form. Again the problem resides in the transformation system. It is not difficult to solve by hand the equation system in the generated time bound program, but the implemented system was not able to do it.

Conclusion.

For all the programs the generated time bound functions have computed total and well approximating time bounds. There is a little overhead in at least the last example, but for the other examples the time bounds are the maximal running times of the programs. The time bounds are found very fast for the first three examples, but a better program transformation system is needed to make the last two time bound programs in closed form. The most time consuming part of the analysis is the data flow analysis. It could be interesting to see whether it is possible to get the same information with a simpler domain such that the analysis will iterate to a fixpoint faster.

8.2 Description of the system

Most part of the system are straightforward implementations of the algorithm described in previous chapters. The system is divided in 8 parts

1. Make step counting version (`timefct reduce1`).
2. Construct Time bound program in $\tilde{\mathbf{L}}$ semantics (`timebound funknavn`).
3. Transform to restricted syntax for 'cons' expressions (`foldout`).
4. Make the data flow analysis (`reynolds`).
5. Compile the set annotated program to standard semantics (`vos-v-comp1`).
6. Foldback from the restricted syntax by unfolding functions not calling them selves (`foldout`).
7. Optimizations.
 - Remove non-recursive variables (`arg-reduce`).
 - Move constants and identical terms out of 'max' operations (`max-reduce`).
 - Remove uncalled functions (`funknavn`).
 - Fold identical definitions (`diff-reduce`).
 - Make local simplifications (`reduce`).
 - Solve finite-difference equations (`diff-eq`).
8. Optimizations.
 - Driving (`drive`).
 - Remove uncalled functions (`funknavn`).

Part 1 is an implementation of the translation scheme in section 3.3. Part 2 composes the program with the inverted length function as described in section 4.7. Part 3 transforms the program to the restricted syntax shown in section 5.0.

Part 4 performs the data flow analysis described in section 5.4. The iteration after fixpoint is done in by repeating a naive depth first evaluation. This strategy is much more efficient than the direct implementation of the algorithm from section 5.4. Part 5 uses the result of the data flow analysis in the compilation to standard semantics. This is an implementation of the method sketched in section 5.6.

The last parts improve the time bound programs in different ways. The central part is the driving algorithm described in chapter 6. In driving expressions we solve or optimize finite-difference equations as it is shown in chapter 7.

To make the driving more efficient some trivial optimizations are performed:

A non-recursive variable is a variable not changed in recursive calls, or not used out of calls. Such variables can safely be removed from the programs. This optimization is also done in Wegbreit's system [Wegbreit, 1975].

Constants and identical terms are moved out of 'max' operations. It is always safe to move identical terms out of the operation, and it will spare some recomputations. We move also constants out as it is safe for a time bound program. We have not examined the importance of this last optimization.

Finally we remove all functions that cannot be reached from the first function in the program.

Chapter 9

Conclusion

We have formalized a construction algorithm for time bound programs. The generated programs have been proven correct and practice shows that they compute total time bound functions for a large class of programs. We have developed a data flow analysis to describe sets of partly known structures, and hereby we approximate a collecting semantics for non-flat domains. Further, we have developed a transformation system to improve time bound programs. The system can solve or improve a number of finite-difference equations.

Compared with the work of Wegbreit [Wegbreit, 1975], we have extended the set of analyzable programs. If a program is non-analyzable it now means that the produced program will not be total instead of letting the analyzer program loop. A main advantage with our system is that we separate the analysis in two phases: time bound construction and program optimization. The first part is specific to the problem, but the second is general as far as any general program transformer can be used.

The two other works [Metayer, 1985] and [Kasai et al., 1980] are heavily based on heuristics. Metayer uses a database of transformation rules, including certain *approximating* transformations. By extending the database with new rules the analyzer is able to handle a richer class of programs. It is not clear how general the rules are and whether the *approximating* transformations must satisfy certain conditions.

The work by Kasai et al. uses annotation in the program. They use an imperative language, and loops can (must?) be annotated with a maximal number of iterations, and conditions should be specified with probabilities for the predicate to be true or false.

The present system uses only a simple sort of annotation in the first part: specification of the inverted length function. The transformation system is still not formalized. A better transformation package will considerably improve the applicability of the system, and it could be interesting to use a full partial evaluator based on composing functions.

Acknowledgements.

Thanks to the MIX-group for a fruitful research environment, to the DIKU night team for the social environment, to Olivier Danvy for his friendly environment and rereading, and to Neil Jones for – everything.

Copenhagen, November 28, 1986

Mads Rosendahl

Chapter 10

Bibliography

[Aho & Johnson, 1974]

A. V. Aho & S. C. Johnson
LR Parsing
Computing Surveys vol. 6, No 2. June, 1974

[Bird, 1976]

Richard Bird
Programs & Machines
John Wiley & Sons, Ltd., 1976.

[Burn, Hankin and Abrahamsky, 1986]

G. L. Burn, C. L.C HaBnkin and S. Abrahamsky
The Theory of Strictness Analysis for Higher Order Functions
Lecture Notes in Computer Science 217, pp. 42 - 62, 1986
Springer Verlag.

[Burstall and Darlington, 1977]

R. M. Burstall and John Darlington
A Transformation System for Developing Recursive Programs
J.ACM vol. 24, no 1, pp. 44 - 67, Jan., 1977.

[Cohen, 1982]

Jacques Cohen
Computer-Assisted Microanalysis of Programs
C.ACM vol. 25, no. 10, pp. 724 - 733, October, 1982

[Cohen and Katcoff, 1977]

Jacques Cohen and J. Katcoff
Symbolic Solution of Finite-Difference Equations
Transactions on Mathematical Software
vol. 3, no. 3, pp. 261 - 271, September, 1977.

[Cousot and Cousot, 1977]

Patrick Cousot and Radhia Cousot
Abstract Interpretation: A Unified Lattice Model for Static Analysis of Pro-

grams by Construction of Approximation of Fixpoints
Fourth Symposium on Principles of Programming Language.
ACM pp. 238 - 252, Jan., 1977.

[Cousot, 1979]

Patrick Cousot
Semantic Foundation of Program Analysis
in: Program Flow Analysis: Theory and Applications. Ed. S. S. Muchnick
and N. D. Jones
Prentice-Hall, pp. 380 - 393, 1981

[Foderaro et al., 1983]

John K. Foderaro, Keith L. Sklower, Kewin Layer
The Franz Lisp manual
University of California, 1983.

[Gordon, 1979]

Michael J. C. Gordon
The Denotational Description of Programming Languages
Springer Verlag, 1979

[Jones, 1987]

Neil D. Jones
Flow Analysis of Lazy Higher Order Functional Programs

[Jones and Mycroft, 1986]

Neil D. Jones and Alan Mycroft
Data Flow Analysis of Applicative Programs using Minimal Function Graphs
13'th Symposium on Principles of Programming Language.
ACM January, 1986.

[Kasai et al, 1980]

Kasai et al.
An Automatic Time Analysis System
Kyoto University, RIMS-335

[Metayer, 1985]

D. le Metayer
Mechanical Analysis of Program Complexity
ACM SIGPLAN 85 Symposium
SIGPLAN vol. 20, no. 7, pp. 69 - 73, July, 1975.

[Mycroft and Nielson, 1983]

Alan Mycroft and Flemming Nielson
Strong Abstract Interpretation using Power Domains
Lecture Notes in Computer Science no. 154
Springer-Verlag, pp. 536 - 547, 1983

[Nielson, 1986]

Flemming Nielson
Expected Forms of Data Flow Analysis
Lecture Notes in Computer Science no. 217
Springer-Verlag, pp. 172 -, 191, 1986

[Nielson, 1985]

Hanne Riis Nielson
Hoare-Like Proof Systems for Run-Time Analysis of Programs
Institute of Electronic Systems
Aalborg University Centre, January, 1985

[Papadimitriou, 1982]

Christos H. Papadimitriou
Combinatorial Optimization: Algorithms and Complexity
Prentice Hall, inc., 1982

[Reynolds, 1969]

John C. Reynolds
Automatic Computation of Data Set Definitions
Information Processing 68, pp. 456 - 461, 1969.

[Sarkar & Hennessy, 1986]

Vivek Sarkar and John Hennessy
Compile-time Partitioning and Scheduling of Parallel Programs
Proc. Sigplan '86 Symp. on Compiler Construction.
Sigplan Notices vol. 21, No. 7, pp. 17 - 26, July, 1986.

[Scherlis, 1981]

William L. Scherlis
Program Improvement by Internal Specialization
8'th Symposium on Principles of Programming Language.
ACM pp. 41 - 49, Jan., 1981.

[Sestoft, 1986]

Peter Sestoft
The structure of a Self-applicable Partial Evaluator
Lecture Notes in Computer Science 217, pp. 236 - 256, 1986
Springer Verlag.

[Sestoft and Søndergaard, 1986]

Peter Sestoft and Harald Søndergaard
Non-Determinacy and its Semantics
Working Paper from ESPRIT project no. 315
Dansk Datamatik Center, 63 pages, 1986

- [Spiegel, 1968]
Murray R. Spiegel
Mathematical Handbook
Mc Graw-Hill, Ltd., 1968
- [Talcott, 1986]
Carolyn Talcott
Derived properties and derived programs
Stanford University May, 1986
- [Tarski, 1955]
Alfred Tarski
A lattice theoretical fixpoint theorem and its applications.
Pacific journal of Math. 5, pp. 285 - 309, June, 1955.
- [Turchin et al, 1982]
V. F. Turchin, R. M. Nirenburg, D. V. Turchin
Experiments with a Supercompiler
ACM Symposium on Lisp and Functional Programming.
Pittsburgh PA, August, 1982.
- [Turchin, 1986]
Valentin F. Turchin
The Concept of a Supercompiler
ACM ToPLaS, Vol. 8, No. 3, pp. 292-325, July, 1986.
- [Wadler, 1984]
Philip Wadler
Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection
on Compile-time
ACM Symposium on Lisp and Functional Programming.
Austin Texas, August, 1984.
- [Wadsworth, 1978]
C. Wadsworth
Lecture Notes in Domain Theory
- [Wegbreit, 1975]
Ben Wegbreit
Mechanical Program Analysis
C.ACM vol. 18, pp. 528 - 539, Sept., 1975.
- [Wegbreit, 1976]
Ben Wegbreit
Goal-Directed Program Transformation
IEEE Transaction on Software Engineering,
vol SE-2, no 2, pp. 69 - 80, June, 1976

[Wegbreit, 1976a]

Ben Wegbreit

Verifying Program Performance

J.ACM. Vol. 23, No. 4, pp. 691 - 699, October, 1976

Appendix A

Output from the analyser system

- 1. Lookup
- 2. Union
- 3. Reverse
- 4. Parse
- 5. Bubble sort

A.1 Lookup

Franz Lisp, Opus 38.69

->

Time used : 3:680 ms (4 garbage collections : 4:600 ms)

Program :

```
( (lookup (x y)
  (if (null y) nil (if (eq (car (car y)) x) (car y)
    (call lookup x (cdr y))))))
```

Inverted length functions :

```
( (all-1 (x) all)
  (length-1 (x) (if (eq x 0) nil (cons all (call length-1 (sub x 1))))))
```

Result : ((time (x y) (add 4 (mul 13 y))))

Time used : 7:120 ms (5 garbage collections : 5:160 ms)

A.2 Union

Franz Lisp, Opus 38.69

->

Time used : 3:940 ms (4 garbage collections : 4:600 ms)

Program :

```
( (union (u v)
  (if (null u)
      v
      (if (call member (car u) v)
          (call union (cdr u) v)
          (cons (car u) (call union (cdr u) v))))))
(member (x s)
  (if (null s) false (if (eq x (car s)) true
                        (call member x (cdr s)))))
```

Inverted length functions :

```
( (length-1 (x) (if (eq x 0) nil
                  (cons all (call length-1 (sub x 1)))))
  (length-1 (x) (if (eq x 0) nil
                  (cons all (call length-1 (sub x 1)))))
```

Result : ((time (u v) (add 4 (mul (add 19 (mul 12 v)) u))))

Time used : 11:780 ms (6 garbage collections : 5:760 ms)

A.3 Reverse

Franz Lisp, Opus 38.69

->

Time used : 3:860 ms (4 garbage collections : 4:620 ms)

Program :

```
( (reverse (x)
  (if (null x) nil (call append (call reverse (cdr x))
                                (cons (car x) nil))))
  (append (x y) (if (null x) y
                    (cons (car x) (call append (cdr x) y)))))
```

Inverted length functions :

```
((length-1 (x) (if (eq x 0) nil
                  (cons all (call length-1 (sub x 1)))))
```

Result : ((time (x) (add 4 (add (mul 10 x) (mul 5 (mul x x)))))

Time used : 18:440 ms (8 garbage collections : 7:020 ms)

A.4 Parse

Franz Lisp, Opus 38.69

->

Time used : 4:060 ms (4 garbage collections : 4:620 ms)

Program :

```
( (parse (inddata) (call main inddata (cons 1 nil) nil))
  (main (in st as)
    (if (eq '1 (car st))
      (if (null in)
        nil
        (if (eq (car in) '-')
          (call main (cdr in) (cons 3 st) (cons (car in) as))
          (call main (cdr in) (cons 7 st) (cons (car in) as))))
      (if (eq '2 (car st))
        (car as)
        (if (eq '3 (car st))
          (if (null in)
            'error
            (call main (cdr in) (cons 7 st) (cons (car in) as)))
          (if (eq '4 (car st))
            (if (null in)
              (call main in (cons 2 (cdr st)) as)
              (call main (cdr in) (cons 9 st) (cons (car in) as)))
            (if (eq '5 (car st))
              (if (null in)
                (if (eq (car (cdr st)) 3)
                  (call main in (cons 8 (cdr st)) as)
                  (call main in (cons 4 (cdr st)) as))
                (if (eq (car in) '*')
                  (call main (cdr in) (cons 10 st) (cons (car in) as))
                  (if (eq (car (cdr st)) 3)
                    (call main in (cons 8 (cdr st)) as)
                    (call main in (cons 4 (cdr st)) as))))
              (if (eq '6 (car st))
                (if (eq (car (cdr st)) 9)
                  (call main in (cons 11 (cdr st)) as)
                  (call main in (cons 5 (cdr st)) as))
                (if (eq '7 (car st))
                  (if (eq (car (cdr st)) 10)
                    (call main in (cons 12 (cdr st)) as)
                    (call main in (cons 6 (cdr st)) as))
                  (if (eq '8 (car st))
                    (if (null in)
                      (call main
```

```

in
  (cons 2 (cdr (cdr st)))
  (cons (cons (car (cdr as)) (cons (car as) nil))
        (cdr (cdr as))))
(call main (cdr in) (cons 9 st)
           (cons (car in) as))
(if (eq '9 (car st))
    (if (null in)
        'error
        (call main (cdr in) (cons 7 st)
                   (cons (car in) as)))
    (if (eq '10 (car st))
        (if (null in)
            'error
            (call main (cdr in) (cons 7 st)
                       (cons (car in) as)))
        (if (eq '11 (car st))
            (if (null in)
                (if (eq (car (cdr (cdr (cdr st)))) 3)
                    (call main
                        in
                        (cons 8 (cdr (cdr (cdr st))))
                        ( cons
                          (cons (car (cdr as))
                                (cons (car (cdr (cdr as)))
                                      (cons (car as) nil)))
                          (cdr (cdr (cdr as))))))
                    (call main
                        in
                        (cons 4 (cdr (cdr (cdr st))))
                        ( cons
                          (cons (car (cdr as))
                                (cons (car (cdr (cdr as)))
                                      (cons (car as) nil)))
                          (cdr (cdr (cdr as))))))
                    (if (eq (car in) '*)
                        (call main
                            (cdr in)
                            (cons 10 st)
                            (cons (car in) as))
                        (if (eq (car (cdr st)) 3)
                            (call main
                                in
                                (cons 8 (cdr (cdr (cdr st))))
                                ( cons

```

```

        (cons (car (cdr as))
              (cons (car (cdr (cdr as)))
                    (cons (car as) nil)))
        (cdr (cdr (cdr as))))))
(call main
 in
 (cons 4 (cdr (cdr (cdr st))))
 ( cons
  (cons (car (cdr as))
        (cons (car (cdr (cdr as)))
              (cons (car as) nil)))
        (cdr (cdr (cdr as))))))
(if (eq '12 (car st))
    (if (eq (car (cdr (cdr (cdr st)))) 9)
        (call main
 in
 (cons 11 (cdr (cdr (cdr st))))
 ( cons
  (cons (car (cdr as))
        (cons (car (cdr (cdr as)))
              (cons (car as) nil)))
        (cdr (cdr (cdr as))))))
    (call main
 in
 (cons 5 (cdr (cdr (cdr st))))
 ( cons
  (cons (car (cdr as))
        (cons (car (cdr (cdr as)))
              (cons (car as) nil)))
        (cdr (cdr (cdr as))))))
    nil)))))))))

```

Inverted length functions :

```

((length-1 (x) (if (eq x 0) nil
                  (cons all (call length-1 (sub x 1))))))

```

Result :

```

( (time (inndata) (add 6 (call t-main--131 inndata '(1))))
  (t-main--131 (x--128 x--129)
    (if (eq (car x--129) 1)
        (if (eq x--128 0)
            9
            (add 23
              (max (call t-main--131 (sub x--128 1) (cons 3 x--129))

```

```

      (call t-main--131 (sub x--128 1) (cons 7 x--129))))))
(if (eq (car x--129) 2)
    12
  (if (eq (car x--129) 3)
      (if (eq x--128 0)
          19
        (add 28 (call t-main--131 (sub x--128 1)
                                (cons 7 x--129))))))
  (if (eq (car x--129) 4)
      (if (eq x--128 0)
          42
        (add 33 (call t-main--131 (sub x--128 1)
                                (cons 9 x--129))))))
  (if (eq (car x--129) 5)
      (if (eq x--128 0)
          (if (eq (car (cdr x--129)) 3) 115 83)
          (if (eq (car (cdr x--129)) 3)
              (add 43
                  (max (call t-main--131 (sub x--128 1)
                                        (cons 10 x--129))
                      (add 99
                          (call t-main--131
                                (sub x--128 1)
                                (cons 9 (cons 8 (cdr x--129))))))))))
          (add 43
              (max (call t-main--131 (sub x--128 1)
                                    (cons 10 x--129))
                  (add 79
                      (call t-main--131
                            (sub x--128 1)
                            (cons 9 (cons 4 (cdr x--129))))))))))
      (if (eq (car x--129) 6)
          (if (eq (car (cdr x--129)) 9)
              (add 43 (call t-main--131 x--128
                                  (cons 11 (cdr x--129))))
            (add 43 (call t-main--131 x--128
                                  (cons 5 (cdr x--129))))))
          (if (eq (car x--129) 7)
              (if (eq (car (cdr x--129)) 10)
                  (add 48 (call t-main--131 x--128
                                          (cons 12 (cdr x--129))))
                (add 48 (call t-main--131 x--128
                                          (cons 6 (cdr x--129))))))
              (if (eq (car x--129) 8)
                  (if (eq x--128 0)

```

```

74
  (add 53
    (call t-main--131 (sub x--128 1)
      (cons 9 x--129))))
(if (eq (car x--129) 9)
  (if (eq x--128 0)
    49
    (add 58
      (call t-main--131 (sub x--128 1)
        (cons 7 x--129))))
  (if (eq (car x--129) 10)
    (if (eq x--128 0)
      54
      (add 63
        (call t-main--131 (sub x--128 1)
          (cons 7 x--129))))
      (if (eq (car x--129) 11)
        (if (eq x--128 0)
          (if (eq (car (cdr (cdr (cdr x--129)))) 3)
            166 134)
          (if (eq (car (cdr x--129)) 3)
            (add 73
              ( max
                (call t-main--131
                  (sub x--128 1)
                  (cons 10 x--129))
                (add 148
                  (call t-main--131
                    (sub x--128 1)
                    (cons 9
                      (cons 8
                        (cdr (cdr (cdr x--129))))))))))))
            (add 73
              ( max
                (call t-main--131
                  (sub x--128 1)
                  (cons 10 x--129))
                (add 128
                  (call t-main--131
                    (sub x--128 1)
                    (cons 9
                      (cons 4
                        (cdr (cdr (cdr x--129))))))))))))
            (if (eq (car x--129) 12)
              (if (eq (car (cdr (cdr (cdr x--129)))) 9)

```

```
(add 94
  (call t-main--131
    x--128
    (cons 11 (cdr (cdr (cdr x--129))))))
(add 94
  (call t-main--131
    x--128
    (cons 5 (cdr (cdr (cdr x--129))))))
61)))))))))))))
```

Time used : 38:15:500 ms (78 garbage collections : 1:40:320 ms)

A.5 Sort

Franz Lisp, Opus 38.69

->

Time used : 4:160 ms (4 garbage collections : 4:740 ms)

Program :

```
( (boble (a) (call sort a (call length a) (call length a)))
  (length (e) (if (null e) 0 (add 1 (call length (cdr e)))))
  (sort (a i j)
    (if (eq i 0) a (if (eq i 1) a
      (call sort (call sort1 a j) (sub i 1) j))))
  (sort1 (a j)
    (if (eq j 0)
      a
      (if (eq j 1)
        a
        (if (lt (call inx a j) (call inx a (sub j 1)))
          (call sort1
            (call chg
              (call chg a j (call inx a (sub j 1)))
              (sub j 1)
              (call inx a j))
            (sub j 1)
            (call sort1 a (sub j 1))))))
    (inx (l i) (if (eq i 1) (car l)
      (call inx (cdr l) (sub i 1))))
    (chg (l i v)
      (if (eq i 1)
        (cons v (cdr l))
        (cons (car l) (call chg (cdr l) (sub i 1) v)))))
```

Inverted length functions :

```
((length-1 (x) (if (eq x 0) nil
  (cons all (call length-1 (sub x 1)))))
```

Result :

```
( (time (a) (add 6 (add (mul 2 (add 4 (mul 8 a))) (call t-sort a a)))
  (t-sort (i j)
    (if (eq i 0)
      5
      (if (eq i 1) 9
        (add 16 (add (call t-sort1 j) (call t-sort (sub i 1) j))))))
  (t-sort1 (j)
    (if (eq j 0)
```

```
5
(if (eq j 1)
  9
  (add 37
    (add (mul 2 (add -4 (mul 10 j)))
      (add (mul 2 (add -4 (mul 10 (sub j 1))))
        (add (call t-sort1 (sub j 1))
          (add (add -6 (mul 14 (sub j 1)))
            (add -6 (mul 14 j))))))))))
```

Time used : 46:480 ms (15 garbage collections : 12:500 ms)