Optimising the Memory Management of Higher–Order Functional Programs

Von der Mathematisch–Naturwissenschaftlichen Fakultät der Rheinisch–Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von
Diplom–Informatiker Markus Mohnen
aus Erkelenz

Referent: Universitätsprofessor Dr. rer. nat. K. Indermark
Korreferent: Prof. dr. ir. M. J. Plasmeijer
Tag der mündlichen Prüfung: 24.11.1997

D 82 (Diss. RWTH Aachen)

Markus Mohnen
Lehrstuhl für Informatik II
Email: mohnen@informatik.rwth-aachen.de

Aachener Informatik–Bericht 97–13

Herausgeber: Fachgruppe Informatik
Ahornstr. 55
RWTH Aachen
52056 Aachen
GERMANY

Druck: Lönnies & Schmitz Druckerei
Warmweiherstr. 22
52066 Aachen
GERMANY

ISSN 0935–3232

Contents

Basic Notations

$\mathbb{B}$	Boolean values $\{0, 1\}$
$\mathbb{N}$	Natural numbers $\{0, 1, 2, \ldots\}$
$\mathfrak{P}(A)$	Set of all subsets of A
$\mathfrak{P}_{\text{fin}}(A)$	Set of all finite subsets of A
$A \uplus B$	Disjoint union of sets A and B
$\langle A^i \mid i \in I \rangle$	Family of sets A^i with index set I (see Appendix A)
$f : A \dashrightarrow B$	Partial function
$\mathsf{Dom}(f)$	Domain of partial function f
$f \cup g$	Union of functions f and g with $\mathsf{Dom}(f) \cap \mathsf{Dom}(g) = \emptyset$: $(f \cup g)(a) := \begin{cases} f(a) & \text{if } a \in \mathsf{Dom}(f) \\ g(a) & \text{if } a \in \mathsf{Dom}(g) \end{cases}$
id	Identity function
undef	Function with empty domain, i.e. $\mathsf{Dom}(\mathsf{undef}) = \emptyset$
$f[a/b]$	Function g resulting from modification of f at a: $g(a') := \begin{cases} b & \text{if } a' = a \\ f(a') & \text{otherwise} \end{cases}$
$f[a_1/b_1, \ldots, a_n, b_n]$	Short form of $f[a_1/b_1] \ldots [a_n/b_n]$
$[a_1/b_1, \ldots, a_n, b_n]$	Short form of $\mathsf{undef}[a_1/b_1, \ldots, a_n, b_n]$
$[A_1 \times \cdots \times A_n \to B]$	Set of all functions $f : A_1 \times \cdots \times A_n \to B$
$(a)_i$	Selection of the i–th component a_i of vector $a = (a_1, \ldots, a_n)$
$\langle A, \leq \rangle$	Partially ordered set (pos) (see Appendix A)
$\langle A_1, \leq_1 \rangle \times \langle A_2, \leq_2 \rangle$	Product of pos (see Appendix A)
$[\langle A_1, \leq \rangle \to \langle A_2, \leq_2 \rangle]$	Function space of pos (see Appendix A)

1. Introduction

In traditional programming languages like Pascal, C, or Ada *recursive dynamic data structures* like lists or trees are available by *explicit* access to the heap of the runtime system. Commands like C's `malloc` or Ada's `new` allocate memory in the heap and provide subsequent access to the allocated memory by *pointers*. As execution of the program proceeds, allocated memory may become obsolete and therefore the *explicit deallocation* of memory is possible by commands like `free` or `UNCHECKED_DEALLOCATION`.

Explicit deallocation of heap allocated objects is a very dangerous concept. If a heap object is deallocated too early, then subsequent access may result in runtime errors. Even if the variable containing the pointer is not used after the deallocation, such runtime errors cannot be avoided: Before the deallocation the pointer to an object may be stored in more than one variable (the object is *aliased*). Hence, after deallocation via one of the variables, the other variables contain *dangling references* to the memory area where the object has been. Besides potential runtime errors caused by access to a free memory area, even worse situations can occur: If the memory area is reused by another allocation, access via the dangling references causes unpredictable program execution. Moreover, the presence of such errors in a program, being a non–trivial program property, is not decidable at compile–time.

The only chance to avoid such erroneous programs is to *disallow* explicit deallocation: This *implicit* memory management guarantees that programs cannot go wrong because of dangling references. Imperative languages featuring implicit memory management are Smalltalk and, more recently, Java. The latter was designed to be a robust and secure language in the first place.

Functional programming languages like Clean or Haskell have an even more implicit memory management, since pointer types are not included in the languages. Instead, dynamic data structures are provided by *algebraic data types*: The structure is described by a finite set of *constructors*. In Haskell, for instance, lists of integers can be defined by: `data [Integer] = [] | Integer:[Integer]`. A value of type `[Integer]` is either the value `[]` (indicating the empty list) or is equal to n:l, where n is a value of type `Integer` and l is another list. Thus, the constructors can be used in the program to create new values. To implement algebraic types, a technique known as *programmed graph reduction* is used: Like in traditional languages a heap is used to store dynamic objects, which are referenced by (implicit) pointers, and the constructors act as allocation instructions.

However, the ban of explicit deallocation has the disadvantage that even objects which are not longer accessible by the program can not be freed. Consequently, *memory leaks* can occur, where the heap is full of such objects. The extremal situation is when a program's execution terminates with a runtime error because memory is exhausted.

To circumvent memory leaks, the runtime system of languages with implicit memory management uses *garbage collection* to find and deallocate inaccessible objects. If the heap is exhausted the actual execution of the program is interrupted and a *garbage collection cycle* is performed: The heap is examined for inaccessible objects and the *garbage* found during the examination is deallocated. After the end of the cycle normal execution is resumed.

This approach has two disadvantages: Firstly, the additional time needed to perform garbage collection cycles increases the overall execution time of the program. Secondly, garbage may exist over a long period of time and hence unnecessarily increase the memory usage. This aspect is especially unfavourable in the context of multitasking environments where other processes may fail to allocate system memory caused by an unnecessarily large heap of another process.

To overcome these disadvantages we observe that garbage collection is *program independent*: The collector has no knowledge of the flow of execution. Especially, the collector has no knowledge of the context in which data structures are used by the program. If it is possible to analyse the memory behaviour of programs at *compile–time* then we can use the information for assisting the garbage collector.

In this thesis, we present *escape analysis*, a method for extracting safe approximations of the memory behaviour of higher–order functional programs and verify the correctness of this analysis. Furthermore, we describe how the information gained by escape analysis can be used to improve the program's execution. Finally, we give measurements of the effect on the runtime behaviour of programs both in terms of memory consumption and execution time.

1.1 The Benefits of Escape Analysis

The aim of escape analysis is to extract information about the storage behaviour wrt. graph reduction from the underlying functional program. The intention is to use this information for a better use of memory at runtime.

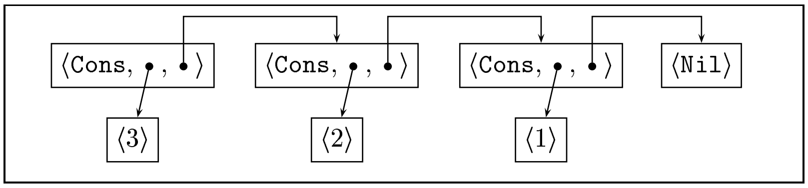
The analysis gives information about those parts of the arguments of a function application which may *escape from the application*. In terms of graph rewriting, those parts which do escape are in the subgraph spanned by the result of the application. For instance, consider the usual definition of the append function:

```
(++) [] l'    = l'
(++) (a:l) l' = a:(l++l')
```

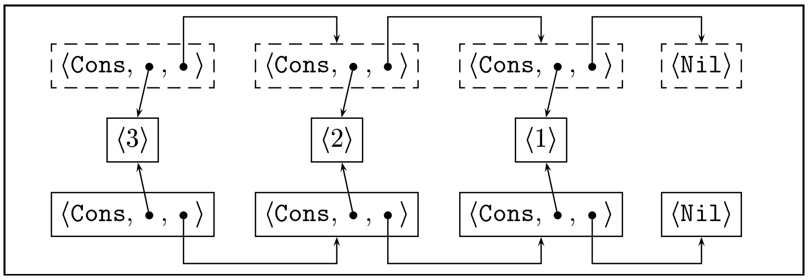
Since the first argument is recursively traversed and copied, the heap cells for its constructors cannot be part of the result. In contrast, the heap cells of the second argument escape.

To clarify this point of view consider the graph representation of the list `[3,2,1]` in Figure 1.1(a). Each entry in the list and each constructor of the list is represented by a separate heap cell. If we evaluate the expression `[3,2,1]++[]` then the result is again `[3,2,1]`. However, the graph (Figure 1.1(b)) now contains garbage cells (represented by a dashed box) for the argument list.

With this knowledge we can modify code generation; if we can ensure that parts which do not escape are unshared, code can be inserted which deallocates these parts after termination of `++`. This optimisation is known as *compile–time garbage collection*.



(a) Graph representation of `[3,2,1]`



(b) Graph after evaluation of `[3,2,1]++[]`

Another application of our analysis can be done in the context of higher-order functions. In the graph reduction approach not only data structures are represented in the graph, but also functional values (closures). This is necessary because λ-abstractions and partial applications allow the creation of new functions at runtime. Since these runtime-created functions may outlive their creating functions it is necessary to choose a representation in a storage area which is not affected by termination of function evaluation, that is in the heap.

In many cases, however, a closure is used only locally. Consider the quicksort function `qs`. Sorting an empty list yields the empty list; for a non–empty list, the first element is used as pivot to split the rest list into two parts, which are then sorted and linked.

```
qs []  = []
qs (a:l) = (qs (filter (<a) l))++[a]++(qs (filter (>=a) l))
```

The partial applications `(<a)` and `(>=a)` do not escape from the applications of `filter`. Hence, the closures can be allocated on the stack. A further analysis of the call structure of `filter` reveals that we can even allocate the closures statically.

1.2 Correctness of Program Analysis

To verify the correctness of methods for analysing programs, we have two basic approaches [HS95a]:

Monolithic: Correctness of the analysis and the transformations based on the results of the analysis are considered simultaneously. The whole process is correct iff the optimised program is equivalent to the original in a chosen model of execution.

Model–based: A semantic property is defined, which is used as an *interface* between analysis and optimisations. The analysis is proved to be a safe approximation of the semantic property. Under the assumption that the property holds, the optimisation is shown to be semantics preserving.

The main advantage of the model–based approach over the monolithic is that the optimisation uses only the semantic property, and no "implementation details" of the analysis. Hence, as long as some other analysis fulfils the same property, we can use the optimisations without having to reprove the correctness of the analysis, and vice versa.

In our case, we separately prove the analysis and the optimisations based on the informations obtained by the analysis to be correct. We show that we can express escaping as a *denotational property* although the notion seems to be operational. Then we prove the analysis to be correct wrt. a (modified) denotational semantics.

1.3 Contributions of the Thesis

This thesis presents a detailed study of escape analysis and its influence on the performance. The main contributions of the thesis are:

- We define the notion of escaping as *denotational* property of programs. This allows us to validate the correctness of the analysis without the necessity to fix the operational model more than marginally. We intentionally avoid the use of abstract machines as operational model, since the implementation details imposed by the choice of an abstract machine are irrelevant for our approach.
- Our escape analysis imposes only a *very small compile–time overhead*. We show that by using special properties of the analysis, we can improve the analysis from exponential to quadratic complexity in the worst–case. For realistic programs, the analysis can be performed at almost linear lime.
- Since the standard theory of denotational semantics, which is based on the fixpoint theorem of Knaster and Tarski, does not allow the definition of a *denotational* model for graph reduction, we develop a suitable extension of the theorem.
- We give the *first model–based proof of correctness* of escape analysis and applications based on escape information.
- Detailed *performance evaluations* allow a precise understanding of the effects of the applications. We show that the combination of traditional and compile–time garbage collection not only improves the memory consumption immensely, but also the runtimes of the programs. The performance evaluations are the most detailed in literature.
- Although the language we use lacks many of the advanced features of modern functional languages, we show that our results can be transfered to realistic languages, demonstrating the *extensibility* of our approach.

Parts of this work have been previously presented in [Moh95a, Moh95b, Moh97].

1.4 Structure of the Thesis

This thesis addresses both theoretical and practical issues concerning escape analysis and we begin by summarising the contents. Following the model–based approach, the thesis is divided into three parts.

In the first part, we define escaping as a semantic property. The functional language F which we introduce in Chapter 2 is simple enough to allow a formal treatment, but also has the potential to serve as core for the advanced features of modern functional languages. Because the standard denotational semantics defined in this chapter is not capable of expressing *escaping*, Chapter 3 describes a conservative extension of the semantics. We show how the semantic domains must be extended to allow the distinction between a value and its copy and how this can be used to formalise escaping. The following chapter (Chapter 4) shows how escaping can be safely approximated by *escape analysis*. Guaranteed termination of the analysis is accomplished by abstract interpretation using finite domains. Moreover, we show that the analysis has a very good worst–case complexity: The number of function evaluations is shown to be quadratic in the size of the program. Chapter 5 concludes the first part with the proof that the escape analysis defined in the preceding chapter is a safe approximation of the augmented semantics.

In the second part, we demonstrate how the knowledge of the escape behaviour of programs can guide optimisations. For this purpose, we define a denotational model of graph reduction in Chapter 6. A problem arises with the definition of the graph domain: The order on the graphs must ignore garbage, because otherwise removal of garbage would be a non–monotonic operation. Consequently, the graph domain is not a partially ordered set, but only a quasi ordered set. This makes the use of the standard theory of denotational semantics impossible. However, by exploiting the special structure of our approach, we manage to find a generalised fixpoint theorem for function spaces of quasi ordered sets. We show that the graph semantics is sound with respect to the reference semantics. Furthermore, escaping in the augmented semantics is shown to be a precise model of reachability from the result in the graph semantics. The two applications discussed in Section 1.1 are formally introduced in Chapter 7 and we prove their correctness. To conclude the second part, experimental results show that a program's memory behaviour is improved by the applications, both in terms of end memory usage and peak usage; in combination with traditional garbage collection, also the runtimes of the programs decrease in most cases.

In the third part, we discuss various extensions. In Chapter 8 we demonstrate extensions of our sample language and their effects on our results. We show how the results for the language can be used as a basis for realistic functional languages. In Chapter 9 we consider related work both for the analysis and program optimisations based on analyses. Finally, Chapter 10 outlines several ideas for further research motivated by the issues raised in the preceding chapters.

Part I: Analysis

2. The Language F

This chapter introduces the language F, which serves as the basis for our investigation. Programs in the language F are *monomorphic higher–order applicative recursive equation systems with data constructors*. While the language is austere, more realistic languages can be easily achieved by adding *a polymorphic type system*, *pattern–matching*, *λ-abstractions*, and *local definitions*[1]. Extensions of F and their influence are discussed in Chapter 8.

2.1 Syntax

The underlying type system distinguishes between *built–in sorts (basic sorts)* and *user–defined sorts (constructed sorts)*, and allows the definition of *functional types*.

Definition 2.1 (Types)
Let $S = BS \uplus CS$ be the set of *sorts*, where BS and CS are finite, disjoint sets of *(basic) sorts* and *constructed sorts* such that there is at least a sort `bool` $\in BS$. The set of *types over S*, denoted by $\mathsf{T}(S)$, is the smallest set where

1. $S \subseteq \mathsf{T}(S)$
2. $t_1, \ldots, t_n \to t \in \mathsf{T}(S)$ if $n \geq 1$, $t, t_1, \ldots, t_n \in \mathsf{T}(S)$ $\triangleleft$

Typically, the type system of higher–order languages allow only unary function types $t' \to t$. This is sufficient since n–ary functions of type $t_1, \ldots, t_n \to t$ can be modelled as $t_1 \to (t_2 \to (\ldots (t_n \to t) \ldots))$ by using *currying* [Sto77]. However, we resign from using currying because it destroys the possibility to distinguish on the syntactic level between partial applications and saturated applications. As we will see later, currying is not possible for the abstract domains: A sequence of n unary applications contains more information than a saturated n–ary application, since the first corresponds to n closures in the graph reduction model, whereas the second corresponds to only one closure.

We assume in the thesis that there are fixed disjoint families of symbols for *variables* $X = \langle X^t \mid t \in \mathsf{T}(S)\rangle$, *defined functions* $DF = \langle DF^{t_1,\ldots,t_n \to t} \mid n \in \mathbb{N}^+, t_1, \ldots, t_n, t \in \mathsf{T}(S)\rangle$, *basic functions* $BF = \langle BF^{bs_1,\ldots,bs_n \to bs} \mid n \in \mathbb{N}^+, bs_1, \ldots, bs_n, bs \in BS\rangle$, and `constructors` $C = \langle C^{s_1,\ldots,s_n \to cs} \mid n \in \mathbb{N}^+, s_1, \ldots, s_n \in S, cs \in CS\rangle$.

[1] In our subsequent examples, however, we already use a syntax with these "syntactic sugarings", i.e., our example programs are essentially in Haskell syntax.

Example: Lists of integers can be represented by the sorts $\texttt{int} \in BS$, $\texttt{ListOfInt} \in CS$ and the constructors $\texttt{Nil} \in C^{\texttt{ListOfInt}}$ and $\texttt{Cons} \in C^{\texttt{int},\texttt{ListOfInt} \to \texttt{ListOfInt}}$. $\diamond$

The restriction of constructors to be first–order is done to avoid the necessity to use *reflexive domains* [Sto77] as semantic domains. We discuss this topic in greater detail in Chapter 8. For the rest of the thesis we assume the families X, DF, BF, and C to be fixed.

The set of values of a sort cs is determined by the set of constructors which have cs as their target sort.

Definition 2.2 (Constructors of sort cs)
The *constructors of sort cs* are defined as $C^{cs} := \{c \in C^{s_1,\ldots,s_n \to cs} \mid n \in \mathbb{N}, s_1, \ldots, s_n \in S\}$. $\triangleleft$

To define recursive programs over constructed values, we need primitives to test and select components.

Definition 2.3 (Constructor Tests, Selectors)
From the constructors we derive

1. the family of *constructor tests* $CTest := \langle CTest^{cs \to \texttt{bool}} \mid cs \in CS \rangle$, where the set of constructor tests of type $cs \to \texttt{bool}$ is defined as $CTest^{cs \to \texttt{bool}} := \{\texttt{is}{-}c \mid c \in C\}$.

2. the family of *selectors* $CSel := \langle CSel^{cs \to s} \mid s \in S \rangle$, where the set of selectors of type $cs \to t$ is defined as $CSel^{cs \to t} := \{\texttt{sel}^j{-}c \mid c \in C^{t_1,\ldots,t_n \to cs}, 1 \le j \le n\colon\ t_j = t\}$. $\triangleleft$

Example: For the list constructors, we obtain the constructor tests $\texttt{is}{-}\texttt{Nil}, \texttt{is}{-}\texttt{Cons} \in CTest^{\texttt{ListOfInt} \to \texttt{bool}}$. Since the $\texttt{Nil}$ constructor has no arguments, it has no associated selectors. For the $\texttt{Cons}$ constructor the selectors are $\texttt{sel}^1{-}\texttt{Cons} \in CSel^{\texttt{ListOfInt} \to \texttt{int}}$ and $\texttt{sel}^2{-}\texttt{Cons} \in CSel^{\texttt{ListOfInt} \to \texttt{ListOfInt}}$. The intended semantics is that they select the head and the tail of the list, respectively. $\diamond$

In more realistic languages *pattern matching* is used instead of these primitives. It is well–know that (sequential) pattern matching can be simulated by these primitives. Alternatively, we could use a `case`–construct for this purpose [PL91a]. However, using this construct would increase the technical complexity of the abstract interpretation.

We collect all intrinsic function symbols in the family $\Omega := \langle \Omega^t \mid t \in \mathsf{T}(S) \rangle$ where the sets Ω^t are defined as $\Omega^t := BF^t \cup C^t \cup CSel^t \cup CTest^t \quad \forall t \in \mathsf{T}(S)$.

We define which expressions are possible as the right hand sides of definitions, which form programs.

Definition 2.4 (Expressions, F-Programs)
We define the family of *expressions* over X, Ω and DF as $E := \langle E^t \mid t \in \mathsf{T}(S) \rangle$, where the sets E^t are defined inductively by

1. $X^t \cup \Omega^t \cup DF^t \subseteq E^t$

2. $(e\ e_1\ \dots\ e_m) \in \begin{cases} E^t & \text{if } m = n \\ E^{t_{m+1},\dots,t_n \to t} & \text{if } 1 \leq m < n \end{cases}$

   if $e \in E^{t_1,\dots,t_n \to t}$, $e_i \in E^{t_i}$ $(1 \leq i \leq m)$

3. $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \in E^t$ if $e \in E^{\texttt{bool}}$ and $e_1, e_2 \in E^t$

An F-*program* is a finite set of *definitions* (one for each defined function)

$$F(x_1, \dots, x_n) := e$$

where $F \in DF^{t_1,\dots,t_n \to t}$, $x_i \in X^{t_i}$ and $e \in E^t$ with variables $\{x_1, \dots, x_n\}$. ◁

Here we see the price we have to pay for abolishing currying: Since we allow partial application, we have to distinguish between partial applications and saturated applications explicitly.

Example: In Haskell syntax, the usual definition of the append–function, which creates the concatenation of two lists, can be formulated in the following way.

```
append [] l'  = l'
append (a:l) l' = a:(append l l')
```

In F syntax, the same function is defined as

```
append(l,l') = if (is–Nil l) then l'
               else (Cons (sel¹–Cons l) (append (sel²–Cons l) l'))
```
◇

2.2 Denotational Semantics

In this section we define the denotational semantics of F which serves as the reference point for all correctness results in this thesis.

All semantics in this thesis are defined in three steps: (1) domains are defined for each type, (2) values from these domains are associated with expressions, and (3) the semantics of programs is defined as a fixpoint.

2.2.1 Denotational Domains

We assume that we have sets V^{bs} for all basic sorts bs, e.g. $V^{\texttt{bool}} = \mathbb{B}$. To model a non–strict semantics in the presence of constructors, we use infinite terms as semantic domains for the constructed sorts. Infinite terms are modelled as ideals of partial terms.

Definition 2.5 (Downward Closed Set, Ideal)
Let $\langle A, \leq \rangle$ be a partial order and $T \subseteq A$ a set. T is called

Downward closed iff for all $a \in T$, $b \in A$ with $b \leq a$ holds that $b \in T$. The *down closure* of T is the smallest downward closed set containing T: $T{\downarrow} := \{b \in A \mid \exists a \in T : \ b \leq a\}$.

Ideal iff T is directed and downward closed. $\lhd$

An ideal is a directed set which has the property that the down closure adds no additional elements.

Starting from a partial order, we can obtain a complete partial order by using *ideal completion.*

LEMMA 2.1 ($\mathsf{Id}(\langle A, \leq\rangle)$)
Let $\langle A, \leq\rangle$ be a po with least element $\perp \in A$.

$$\mathsf{Id}(\langle A, \leq\rangle) := \langle\{T \subseteq A \mid T \text{ ideal}\}, \subseteq\rangle$$

is a complete partial order (cpo) with least element $\{\perp\}$.
$\langle A, \leq\rangle$ is *embedded* in $\mathsf{Id}(\langle A, \leq\rangle)$ by the mapping $a \mapsto \{a' \in A \mid a' \leq a\}$. $\square$

In contrast to the definition of infinite terms by means of directed sets [Thi94] this approach has the advantage that an ideal uniquely determines an infinite term. For directed sets, we would have to define an equivalence relation to obtain an unique infinite term.

DEFINITION 2.6 (SEMANTIC DOMAINS)
For each $t \in \mathsf{T}(S)$ we define the *semantic domain* $\langle \mathsf{CT}^t, \leq^t\rangle$ as follows:

1. $\mathsf{CT}^{bs} := \langle V^{bs} \uplus \{\perp^{bs}\}, \leq^{bs}\rangle$ with $\perp^{bs} \leq^{bs} v$ and $v \leq^{bs} v$ for all $v \in \mathsf{CT}^{bs}$ for all basic sort $bs \in BS$.

2. For all constructed sorts $cs \in CS$ the sets PT^{cs} of *partial terms* are defined as the smallest sets where

   (a) $\perp^{cs} \in \mathsf{PT}^{cs}$

   (b) $c(v_1, \ldots, v_n) \in \mathsf{PT}^{cs}$ if $c \in C^{t_1,\ldots,t_n \to cs}$, $v_i \in \begin{cases} \mathsf{PT}^{t_i} & \text{if } t_i \in CS \\ \mathsf{CT}^{t_i} & \text{otherwise} \end{cases}$ for $1 \leq i \leq n$

   Furthermore, we define relations $\leq_\perp^{cs} \subseteq \mathsf{PT}^{cs} \times \mathsf{PT}^{cs}$ accordingly:

   (a) $\perp^{cs} \leq_\perp^{cs} v$ and $v \leq_\perp^{cs} v$ for all $v \in \mathsf{PT}^{cs}$

   (b) $c(v_1, \ldots, v_n) \leq_\perp^{cs} c(v'_1, \ldots, v'_n)$ for all $c \in C^{t_1,\ldots,t_n \to cs}$, $v_i, v'_i \in \begin{cases} \mathsf{PT}^{t_i} & \text{if } t_i \in CS \\ \mathsf{CT}^{t_i} & \text{otherwise} \end{cases}$ with $v_i \leq_\perp^{t_i} v'_i$ if $t_i \in CS$ or $v_i \leq^{t_i} v'_i$ otherwise for $1 \leq i \leq n$

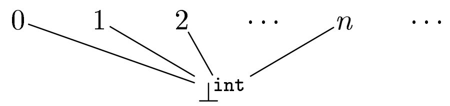
   We define the set of *infinite terms of sort cs* as $\mathsf{CT}^{cs} := \mathsf{Id}(\langle \mathsf{PT}^{cs}, \leq_\perp^{cs}\rangle)$.

3. $\mathsf{CT}^{t_1,\ldots,t_n \to t} := [\langle \mathsf{CT}^{t_1}, \leq^{t_1}\rangle \times \cdots \times \langle \mathsf{CT}^{t_n}, \leq^{t_n}\rangle \to \langle \mathsf{CT}^{t_n}, \leq^{t_n}\rangle]$ for $t_1, \ldots, t_n \to t \in \mathsf{T}(S)$.

In addition, we denote $\mathsf{CT} := \langle \mathsf{CT}^t \mid t \in \mathsf{T}(S)\rangle$ and $\perp^t$ as the least element of CT^t, i.e. $\perp^{bs} := \perp^{bs}$, $\perp^{cs} := \{\perp^{cs}\}$, and $\perp^{t_1,\ldots,t_n \to t} := \lambda(v_1, \ldots, v_n).\perp^t$. $\lhd$

Examples:

1. For basic sorts bs, the semantic domain CT^{bs} is the flat cpo resulting from adding a distinct bottom element $\perp^{bs}$ to the set V^{bs}. For example, for the sort $\mathtt{int} \in BS$ with $V^{\mathtt{int}} = \mathbb{N}$, we obtain the following structure



2. In the presence of non–strict constructors, the semantics domain for constructed sorts must contain

   (a) *Finite terms* for terminating computations, like finite lists $\mathtt{Cons}(1, \mathtt{Nil})$.

   (b) *Partial terms* for computations which create a finite part of the terms and then are non–productive non–terminating. Since the latter is expressed by the semantical value $\perp^{cs}$, we obtain finite terms which may contain such bottom elements. An example for lists is $\mathtt{Cons}(1, \perp^{\mathtt{ListOfInt}})$.

   (c) *Infinite terms* for non–terminating but productive computations, like the infinite list of all natural numbers $\mathtt{Cons}(1, \mathtt{Cons}(2, \mathtt{Cons}(3, \ldots)))$. These values are represented by infinite "sequences" of partial terms, e.g.

   $$\begin{array}{l}\{\perp^{\mathtt{ListOfInt}}, \mathtt{Cons}(1, \perp^{\mathtt{ListOfInt}}), \mathtt{Cons}(\perp^{\mathtt{int}}, \perp^{\mathtt{ListOfInt}}),\\ \ \mathtt{Cons}(1, \mathtt{Cons}(2, \perp^{\mathtt{ListOfInt}})), \ldots\}\end{array}$$

   for the list of natural numbers. To obtain an unique representation, we do not consider sequences, but ideals.

   Finite and partial terms are embedded in the ideal representation by finite ideals, e.g.

   $$\begin{array}{rcl}\mathtt{Cons}(1, \mathtt{Nil}) & \mapsto & \{\perp^{\mathtt{ListOfInt}}, \mathtt{Cons}(\perp^{\mathtt{int}}, \perp^{\mathtt{ListOfInt}}),\\ & & \ \mathtt{Cons}(1, \perp^{\mathtt{ListOfInt}}), \mathtt{Cons}(\perp^{\mathtt{int}}, \mathtt{Nil}), \mathtt{Cons}(1, \mathtt{Nil})\}\\ \mathtt{Cons}(1, \perp^{\mathtt{ListOfInt}}) & \mapsto & \{\perp^{\mathtt{ListOfInt}}, \mathtt{Cons}(\perp^{\mathtt{int}}, \perp^{\mathtt{ListOfInt}}), \mathtt{Cons}(1, \perp^{\mathtt{ListOfInt}})\}\end{array}$$

3. Functional types are modelled as usual: The elements of $\mathsf{CT}^{t_1,\ldots,t_n \to t}$ are functions f of type $f : \mathsf{CT}^{t_1} \times \cdots \times \mathsf{CT}^{t_n} \to \mathsf{CT}^{t}$ and we have $f_1 \leq^{t_1,\ldots,t_n \to t} f_2$ iff for all arguments $x_i \in \mathsf{CT}^{t_i}$ $(1 \leq i \leq n)$ holds: $f(x_1, \ldots, x_n) \leq^{t} f'(x_1, \ldots, x_n)$.

Please note that CT^{cs} would be undefined, if we had allowed constructors to have functional arguments. For instance, assume that we have a sort $cs \in CS$ and a constructor $c \in C^{(cs \to cs) \to cs}$. Corresponding to the above definition, this would lead to $\mathsf{PT}^{cs} = c([\mathsf{PT}^{cs} \to \mathsf{PT}^{cs}])$.

Corollary 2.1
The semantic domains CT are cpos. $\square$

The following lemma guarantees that we can use CT^{t} 'almost' like a set of terms.

Lemma 2.2 (Unique Decomposition)
Let $v \in \mathsf{CT}^{cs}$ with $v \neq \perp^{cs}$. There exist uniquely determined $n \in \mathbb{N}$, $c \in C^{t_1,\ldots,t_n \to cs}$ and $v_i \in \mathsf{CT}^{t_i}$ for $1 \leq i \leq n$ such that

$$v = \{c(v'_1, \ldots, v'_n) \mid v'_i \begin{cases} \in v_i & \text{if } t_i \in CS \\ = v_i & \text{otherwise} \end{cases}, 1 \leq i \leq n\}\!\downarrow$$

We write: $v = c(v_1, \ldots, v_n)$. □

The downward closure is necessary to ensure that the right–hand side is an ideal. Its purpose is only to add the missing element $\perp^{cs}$, i.e.

$$\begin{aligned} & \{c(v'_1, \ldots, v'_n) \mid v'_i \begin{cases} \in v_i & \text{if } t_i \in CS \\ = v_i & \text{otherwise} \end{cases}, 1 \leq i \leq n\}\!\downarrow \\ = \; & \{c(v'_1, \ldots, v'_n) \mid v'_i \begin{cases} \in v_i & \text{if } t_i \in CS \\ = v_i & \text{otherwise} \end{cases}, 1 \leq i \leq n\} \cup \{\perp^{cs}\} \end{aligned}$$

2.2.2 Denotational Semantics

Since expressions contain (free) variables, we use *environments* to give a semantics for these variables.

Definition 2.7 (Environments)
Let $A = \langle A^t \mid t \in \mathsf{T}(S) \rangle$ be a family of variables and $B = \langle B^t \mid t \in \mathsf{T}(S) \rangle$ be a family of values. The *set of environments over A and B* $\mathsf{Env}(A, B)$ is defined as

$$\mathsf{Env}(A, B) := \{\alpha : A \dashrightarrow B \mid \mathsf{Dom}(\alpha) < \infty,\ \alpha(a) \in B^t \text{ iff } a \in A^t \cap \mathsf{Dom}(\alpha), t \in \mathsf{T}(S)\} \quad \lhd$$

We use various lowercase Greek letters to denote different kinds of environments.

Definition 2.8 (Basic Operations, Operations)

1. The family of *basic operations* is defined as $\mathsf{BOps} := \langle \mathsf{BOps}^{t_1,\ldots,t_n \to t} \mid t_1, \ldots, t_n \to t \in \mathsf{T}(S) \rangle$, with $\mathsf{BOps}^{t_1,\ldots,t_n \to t} := [V^{t_1} \times \cdots \times V^{t_n} \to V^t]$.
2. The family of *operations* is defined as $\mathsf{Ops} := \langle \mathsf{Ops}^{t_1,\ldots,t_n \to t} \mid t_1, \ldots, t_n \to t \in \mathsf{T}(S) \rangle$, with $\mathsf{Ops}^{t_1,\ldots,t_n \to t} := \mathsf{CT}^{t_1,\ldots,t_n \to t}$. $\lhd$

In the sequel, we assume that the interpretation of basic functions is fixed by the environment $\varrho \in \mathsf{Env}(BF, \mathsf{BOps})$.

Definition 2.9 (Semantics of Intrinsic Functions $\mathfrak{M}$)
Let $f \in \Omega^{t_1,\ldots,t_n \to t}$ be an intrinsic function symbol. The *semantics of f* $\mathfrak{M}[\![f]\!] : \mathsf{CT}^{t_1} \times \cdots \times \mathsf{CT}^{t_n} \to \mathsf{CT}^t$ is defined as

1. $\mathfrak{M}[\![bf]\!] := \lambda(v_1, \ldots, v_n). \begin{cases} \perp^t & \text{if } \exists 1 \leq j \leq n \text{ with } v_j = \perp^{t_j} \\ \varrho(bf)(v_1, \ldots, v_n) & \text{otherwise} \end{cases}$

   for all $bf \in BF$

2. $\mathfrak{M}[\![c]\!] := \lambda(v_1,\ldots,v_n).c(v_1,\ldots,v_n)$ for all $c \in C$ (see Lemma 2.2)

3. $\mathfrak{M}[\![\texttt{is}-c]\!] := \lambda(v).\begin{cases} \bot^{\texttt{bool}} & \text{if } v = \bot^{cs} \\ \texttt{true} & \text{if } v = c(v_1,\ldots,v_m) \\ \texttt{false} & \text{otherwise} \end{cases}$

   for all $\texttt{is}-c \in CTest$

4. $\mathfrak{M}[\![\texttt{sel}^{\underline{j}}-c]\!] := \lambda(v).\begin{cases} v_j & \text{if } v = c(v_1,\ldots,v_m) \text{ and } 1 \le j \le m \\ \bot^t & \text{otherwise} \end{cases}$

   for all $\texttt{sel}^{\underline{j}}-c \in CSel$ ◁

This definition fixes the intuition we have for constructors and associated functions: The constructor test on a constructed value satisfies $\mathfrak{M}[\![\texttt{is}-c]\!](c'(v)) = \texttt{true}$ iff $c' = c$ and the selection on a constructed value fulfils $\mathfrak{M}[\![\texttt{sel}^{\underline{j}}-c]\!](c(v_1,\ldots,v_n)) = v_j$ for $1 \le j \le n$.

The denotational semantics of an expression is determined by the interpretation of the variables X, the defined functions DF, and the semantics of intrinsic functions. It is defined by induction on the structure of expressions.

Definition 2.10 (Expression Semantics $\mathfrak{M}$)
Let $e \in E^t$ be an expression, $\beta \in \mathsf{Env}(X, \mathsf{CT})$ be an environment for variables, and $\sigma \in \mathsf{Env}(DF, \mathsf{Ops})$ be an environment for defined functions. The *semantics of e under β and σ* ($\mathfrak{M}[\![e]\!](\beta,\sigma) \in \mathsf{CT}^t$) is inductively defined:

- $\mathfrak{M}[\![x]\!](\beta,\sigma) := \beta(x)$ for $x \in X^t$
- $\mathfrak{M}[\![F]\!](\beta,\sigma) := \sigma(F)$ for $F \in DF^t$
- $\mathfrak{M}[\![f]\!](\beta,\sigma) := \mathfrak{M}[\![f]\!]$ for $f \in \Omega^t$
- $\mathfrak{M}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\beta,\sigma) :=$
  $$\begin{cases} f(\mathfrak{M}[\![e_1]\!](\beta,\sigma),\ldots,\mathfrak{M}[\![e_n]\!](\beta,\sigma)) & \text{if } m = n \\ \lambda(v_{m+1},\ldots,v_n).f(\mathfrak{M}[\![e_1]\!](\beta,\sigma),\ldots,\mathfrak{M}[\![e_m]\!](\beta,\sigma),v_{m+1},\ldots,v_n) & \text{if } 1 \le m < n \end{cases}$$
  where $f = \mathfrak{M}[\![e_0]\!](\beta,\sigma)$ for $e_0 \in E^{t_1,\ldots,t_n \to t_0}$ and $e_i \in E^{t_i}$ $(1 \le i \le m)$
- $\mathfrak{M}[\![\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2]\!](\beta,\sigma) := \begin{cases} \mathfrak{M}[\![e_1]\!](\beta,\sigma) & \text{if } \mathfrak{M}[\![e_0]\!](\beta,\sigma) = \texttt{true} \\ \bot^t & \text{if } \mathfrak{M}[\![e_0]\!](\beta,\sigma) = \bot^{\texttt{bool}} \\ \mathfrak{M}[\![e_2]\!](\beta,\sigma) & \text{if } \mathfrak{M}[\![e_0]\!](\beta,\sigma) = \texttt{false} \end{cases}$

  for $e_0 \in E^{\texttt{bool}}$ and $e_1, e_2 \in E^t$ ◁

Again, we have to handle the distinction between saturated applications and partial applications explicitly, which result in a new function.

Example: For the right hand side

```
e = if (is−Nil l) then l’
    else (Cons (sel1−Cons l) (append (sel2−Cons l) l’))
```

of the `append` definition, a variable environment $\beta = [\mathtt{l}/\{\bot^{\mathtt{ListOfInt}}, \mathtt{Nil}\}, \mathtt{l'}/v']$, and any environment σ for defined functions, we get

$$\begin{array}{rcl} \mathfrak{M}[\![e]\!](\beta,\sigma) & = & \begin{cases} \mathfrak{M}[\![\mathtt{l'}]\!](\beta,\sigma) & \text{if } v_1 = \mathtt{true} \\ \bot^t & \text{if } v_1 = \bot^{\mathtt{bool}} \\ \mathfrak{M}[\![\mathtt{(Cons\ \ldots)}]\!](\beta,\sigma) & \text{if } v_1 = \mathtt{false} \end{cases} \\ & & \text{where } \; v_1 \;=\; \mathfrak{M}[\![\mathtt{(is\text{-}Nil\ l)}]\!](\beta,\sigma) \\ & & \qquad\quad\;\; = \; (\mathfrak{M}[\![\mathtt{is\text{-}Nil}]\!](\beta,\sigma))(\mathfrak{M}[\![\mathtt{l}]\!](\beta,\sigma)) \\ & & \qquad\quad\;\; = \; \mathtt{true} \\ & = & \mathfrak{M}[\![\mathtt{l'}]\!](\beta,\sigma) = \beta(\mathtt{l'}) = v' \end{array} \qquad \diamond$$

The denotational semantics of a program is usually identified with the semantics of a main function, typically the first function in the program. Hence, notions like equivalence or correctness are defined solely in terms of the main function with this approach. However, our interest is to give an abstract interpretation which gives correct approximations of escaping for all functions of a program. Therefore, we use a more general approach, where the semantics of a program is an *environment*, which associates a meaning to each function in the program.

Definition 2.11 (Program Semantics $\mathfrak{M}$)
Given a program $P = (F_j(x_{j1}, \ldots, x_{jn_j}) := e_j \mid 1 \le j \le p)$ with $F_j \in DF^{t_{j1},\ldots,t_{jn_j} \to t_j}$, $x_{ji} \in X^{t_{ji}}$, and $e_j \in E^{t_j}$ with variables $\{x_{j1}, \ldots, x_{jn_j}\}$ for $1 \le j \le p$, the *semantics of* P is an environment $\mathfrak{M}[\![P]\!] \in \mathsf{Env}(DF, \mathsf{CT})$ defined as

$$\mathfrak{M}[\![P]\!] := [F_1/\mathsf{fix}(\Phi_{\mathfrak{M},P})_1, \ldots, F_p/\mathsf{fix}(\Phi_{\mathfrak{M},P})_p]$$

Here, $\mathsf{fix}(\Phi_{\mathfrak{M},P})$ is the least fixpoint of the transformation $\Phi_{\mathfrak{M},P} : \mathsf{FS}_{\mathfrak{M},P} \to \mathsf{FS}_{\mathfrak{M},P}$ on the function space

$$\mathsf{FS}_{\mathfrak{M},P} := \prod_{j=1}^{p} \left[\mathsf{CT}^{t_{j1}} \times \cdots \times \mathsf{CT}^{t_{jn_j}} \to \mathsf{CT}^{t_j}\right]$$

The transformation is defined as

$$\Phi_{\mathfrak{M},P}(g_1, \ldots, g_n) \;:=\; \begin{pmatrix} \lambda(v_{11}, \ldots, v_{1n_1}).\mathfrak{M}[\![e_1]\!]([x_{11}/v_{11}, \ldots, x_{1n_1}/v_{1n_1}], \sigma) \\ \vdots \\ \lambda(v_{p1}, \ldots, v_{pn_p}).\mathfrak{M}[\![e_p]\!]([x_{p1}/v_{p1}, \ldots, x_{pn_p}/v_{pn_p}], \sigma) \end{pmatrix}$$
$$\text{where } \sigma := [F_1/g_1, \ldots, F_p/g_p] \qquad \triangleleft$$

Because the semantic domains are cpo's and the expression semantics defines continuous functions, the fixpoint theorem of Knaster and Tarski (Theorem A.1) guarantees that $\mathsf{fix}(\Phi_{\mathfrak{M},P})$ exists and can be represented in the following way:

$$\mathsf{fix}(\Phi_{\mathfrak{M},P}) = \bigsqcup \{\Phi^i_{\mathfrak{M},P}(\bot^{\mathsf{FS}_{\mathfrak{M},P}}) \mid i \in \mathbb{N}\}$$

Hence, $\mathfrak{M}[\![P]\!]$ is well–defined.

Example: For the append program $P_{\texttt{append}}$, we have the following transformation

$$\Phi_{\mathfrak{M},P_{\texttt{append}}}(g) = \lambda(v_1, v_2).\mathfrak{M}[\![\texttt{if (is-Nil l) ...}]\!]([\texttt{l}/v_1, \texttt{l'}/v_2], [\texttt{append}/g])$$

Successive application of $\Phi_{\mathfrak{M},P_{\texttt{append}}}$ on the least element $\perp = \lambda(x,y).\perp^{\texttt{ListOfInt}}$ of the associated function space $\mathsf{FS}_{\mathfrak{M},P_{\texttt{append}}} = \mathsf{CT}^{\texttt{ListOfInt},\texttt{ListOfInt}\to\texttt{ListOfInt}}$ yields the following sequence:

$$\begin{array}{rcl}
\Phi^1_{\mathfrak{M},P_{\texttt{append}}}(\perp) & = & (v_1,v_2) \mapsto \begin{cases} v_2 & \text{if } v_1 = \texttt{Nil} \\ \perp^{\texttt{ListOfInt}} & \text{otherwise} \end{cases} \\
\Phi^2_{\mathfrak{M},P_{\texttt{append}}}(\perp) & = & (v_1,v_2) \mapsto \begin{cases} v_2 & \text{if } v_1 = \texttt{Nil} \\ \texttt{Cons}(n, v_2) & \text{if } v_1 = \texttt{Cons}(n, \texttt{Nil}) \\ \texttt{Cons}(n, \perp^{\texttt{ListOfInt}}) & \text{if } v_1 = \texttt{Cons}(n, \perp^{\texttt{ListOfInt}}) \\ \perp^{\texttt{ListOfInt}} & \text{otherwise} \end{cases} \\
& \vdots & \\
\Phi^m_{\mathfrak{M},P_{\texttt{append}}}(\perp) & = & (v_1,v_2) \mapsto \begin{cases} v_2 & \text{if } v_1 = \texttt{Nil} \\ \texttt{Cons}(n_1, v_2) & \text{if } v_1 = \texttt{Cons}(n_1, \texttt{Nil}) \\ \texttt{Cons}(n, \perp) & \text{if } v_1 = \texttt{Cons}(n, \perp) \\ \vdots & \\ \texttt{Cons}(n_1, \ldots \texttt{Cons}(n_m, v_2)\ldots) & \\ & \text{if } v_1 = \texttt{Cons}(n_1, \ldots \texttt{Cons}(n_m, \texttt{Nil})\ldots) \\ \texttt{Cons}(n_1, \ldots \texttt{Cons}(n_m, \perp)\ldots) & \\ & \text{if } v_1 = \texttt{Cons}(n_1, \ldots \texttt{Cons}(n_m, \perp)\ldots) \\ \perp^{\texttt{ListOfInt}} & \text{otherwise} \end{cases}
\end{array}$$

The least upper bound of this sequence is the function $f_{\texttt{append}} = \mathfrak{M}[\![P_{\texttt{append}}]\!](\texttt{append})$ with

$$f_{\texttt{append}} : (v_1, v_2) \mapsto \begin{cases} v_2 & \text{if } v_1 = \texttt{Nil} \\ \texttt{Cons}(n_1, \ldots \texttt{Cons}(n_m, v_2)\ldots) & \\ & \text{if } v_1 = \texttt{Cons}(n_1, \ldots \texttt{Cons}(n_m, \texttt{Nil})\ldots) \\ v_1 & \text{otherwise} \end{cases}$$

For finite lists v_1, $f_{\texttt{append}}(v_1, v_2)$ is the concatenation of v_1 and v_2. It is partial or infinite iff v_2 is partial or infinite. For partial or infinite v_1 we have $f_{\texttt{append}}(v_1, v_2) = v_1$. ◇

2.3 Summary

We have presented syntax and denotational semantics of the language F. Programs are sets of definitions and the semantics of a program is an environment. Expressions are built by basic functions, constructors, constructors test, selectors, application (saturated and partial), and branching.

3. Escaping as Denotational Property

Escaping seems to be a property which can only be expressed operationally: In terms of graph rewriting, those parts which do escape from an application are in the subgraph spanned by the result. To express this notion denotationally, we need the ability to distinguish between 'original' (the argument) and 'copy' (the result). However, this difference cannot be expressed by the denotational semantics $\mathfrak{M}$ we gave in the previous chapter. Consider for instance the `append` function. We want to express that the value of `l` does not escape from the expression $\mathfrak{M}[\![\texttt{(append l [])}]\!](\beta, \mathfrak{M}[\![P]\!])$. But obviously

$$\mathfrak{M}[\![\texttt{(append l [])}]\!](\beta, \mathfrak{M}[\![P]\!]) = \beta(\texttt{l})$$

holds for all environments $\beta \in \mathsf{Env}(X, \mathsf{CT})$. Therefore, we cannot express escaping using the standard semantic domains.

Our solution is to modify the denotational domains by *augmenting* the values with additional tags, which can then be used to express the difference between 'original' and 'copy'. We define a new denotational semantics $\widehat{\mathfrak{M}}$ and show that it behaves like the original semantics $\mathfrak{M}$, except that tags are propagated but without creating non–zero tags.

For the above example, we would consider an augmented environment $\widehat{\beta}$ which assigns an augmented value $\widehat{\beta}(\texttt{l})$ with non–zero tags to `l`, e.g. $\widehat{\beta}(\texttt{l}) = (1, \texttt{Cons}((1,42),(1,\texttt{Nil})))$. The augmented semantics yields

$$\widehat{\mathfrak{M}}[\![\texttt{(append l [])}]\!](\widehat{\beta}, \widehat{\mathfrak{M}}[\![P]\!]) = (0, \texttt{Cons}((1,42),(0,\texttt{Nil}))) \neq \widehat{\beta}(\texttt{l})$$

The augmented domains allow a prediction of the escape behaviour of a graph reduction implementation. With this augmented semantics, we can define escaping as a denotational property.

Special care is taken to ensure that the augmented domains are cpos. Unfortunately, we cannot use the approach we take in Section 6.4, where we develop a fixpoint theory for quasi ordered sets. These structures lack the anti–symmetry of partially ordered sets. Modelling the augmented domains as quasi ordered sets would be straightforward: The added tags would be ignored by the order. In [Moh97], we used this technique to model the problem of dead code elimination in the simple imperative language `while`. However, the notions introduced in Section 6.4 are not suitable for lazy semantics.

3.1 *Augmented Domains*

We obtain augmented domains by adding an extra boolean tag $b \in \mathbb{B}$ to each basic value, each constructor value, and each functional value.

Definition 3.1 (Augmented Semantic Domains)
For all $t \in \mathsf{T}(S)$ we define the *augmented semantic domain* $\langle \widehat{\mathsf{CT}}^t, \widehat{\leq}^t \rangle$

1. $\widehat{\mathsf{CT}}^{bs} := \langle \mathbb{B},\ \leq \rangle \times \langle \mathsf{CT}^{bs},\ \leq^{b} \rangle$ for all $bs \in BS$, where $\leq$ is the usual order with $0 \leq 1$.

2. For all $cs \in CS$ the sets of *augmented partial values* $\widehat{\mathsf{PT}}^{cs}$ are defined as

   (a) $(b, \perp^{cs}) \in \widehat{\mathsf{PT}}^{cs}$ for $b \in \mathbb{B}$

   (b) $(b, c(\widehat{v}_1, \ldots, \widehat{v}_n)) \in \widehat{\mathsf{PT}}^{cs}$ if $c \in C^{t_1,\ldots,t_n \to cs}$, $b \in \mathbb{B}$, and either $\widehat{v}_i \in \widehat{\mathsf{PT}}^{t_i}$ if $t_i \in CS$, or $\widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}$ otherwise $(1 \leq i \leq n)$

   The relations $\widehat{\leq}^{cs}_{\perp} \ \subseteq\ \widehat{\mathsf{PT}}^{cs} \times \widehat{\mathsf{PT}}^{cs}$ are defined as $(b_1, v_1)\ \widehat{\leq}^{cs}_{\perp}\ (b_2, v_2)$ iff $b_1 \leq b_2$ and either $v_1 = \perp^{cs}$ or $v_j = c(\widehat{v}_{j1}, \ldots, \widehat{v}_{jn})$ $(1 \leq j \leq 2)$ where for $1 \leq i \leq n$ holds: either (1) $\widehat{v}_{1i}, \widehat{v}_{2i} \in \widehat{\mathsf{PT}}^{t_i}$ with $\widehat{v}_{1i}\ \widehat{\leq}^{t_i}_{\perp}\ \widehat{v}_{2i}$ if $t_i \in CS$, or (2) $\widehat{v}_{1i}, \widehat{v}_{2i} \in \widehat{\mathsf{CT}}^{t_i}$ with $\widehat{v}_{1i}\ \widehat{\leq}^{t_i}\ \widehat{v}_{2i}$ otherwise.

   We define the sets of *augmented infinite terms* as $\widehat{\mathsf{CT}}^{cs} := \mathsf{Id}(\langle \widehat{\mathsf{PT}}^{cs},\ \widehat{\leq}^{cs}_{\perp} \rangle)$.

3. $\widehat{\mathsf{CT}}^{t_1,\ldots,t_n \to t} := \langle \mathbb{B},\ \leq \rangle \times [\langle \widehat{\mathsf{CT}}^{t_1},\ \widehat{\leq}^{t_1} \rangle \times \cdots \times \langle \widehat{\mathsf{CT}}^{t_n},\ \widehat{\leq}^{t_n} \rangle \to \langle \widehat{\mathsf{CT}}^{t},\ \widehat{\leq}^{t} \rangle]$ for $t_1, \ldots, t_n \to t \in \mathsf{T}(S)$.

In addition, we define the family $\widehat{\mathsf{CT}} := \langle \widehat{\mathsf{CT}}^{t} \mid t \in \mathsf{T}(S) \rangle$ and $\widehat{\perp}^{t}$ as the least element of $\widehat{\mathsf{CT}}^{t}$, i.e. $\widehat{\perp}^{bs} := (0, \perp^{bs})$, $\widehat{\perp}^{cs} := \{(0, \perp^{cs})\}$, and $\widehat{\perp}^{t_1,\ldots,t_n \to t} := (0, \lambda(\widehat{v}_1, \ldots, \widehat{v}_n).\widehat{\perp}^{t})$. $\lhd$

Remarks:

- Figure 3.1 shows some examples for augmented domains.

- Tags are added where separate graph nodes in the execution model are used.

- Every directed set in CT^{t} has several augmented counterparts in $\widehat{\mathsf{CT}}^{t}$, which are also directed sets. Hence the computational structure is preserved.

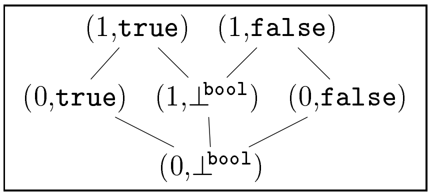
- Currying is not possible any more. Consider the type $t_1, t_2 \to t$ and its curried counterpart $t_1 \to (t_2 \to t)$. The set $\widehat{\mathsf{CT}}^{t_1,t_2 \to t} = \mathbb{B} \times [\widehat{\mathsf{CT}}^{t_1} \times \widehat{\mathsf{CT}}^{t_2} \to \widehat{\mathsf{CT}}^{t}]$ is obviously not isomorphic to the set $\widehat{\mathsf{CT}}^{t_1 \to (t_2 \to t)} = \mathbb{B} \times [\widehat{\mathsf{CT}}^{t_1} \to (\mathbb{B} \times [\widehat{\mathsf{CT}}^{t_2} \to \widehat{\mathsf{CT}}^{t}])]$. The latter contains more information since it has a tag for the result of a partial application of a value of type t_1.

- In the remainder of the thesis we introduce several functions (Definitions 3.2 and 5.1, ...) which operate on $\widehat{\mathsf{CT}}$. We define such a function f in the following way:

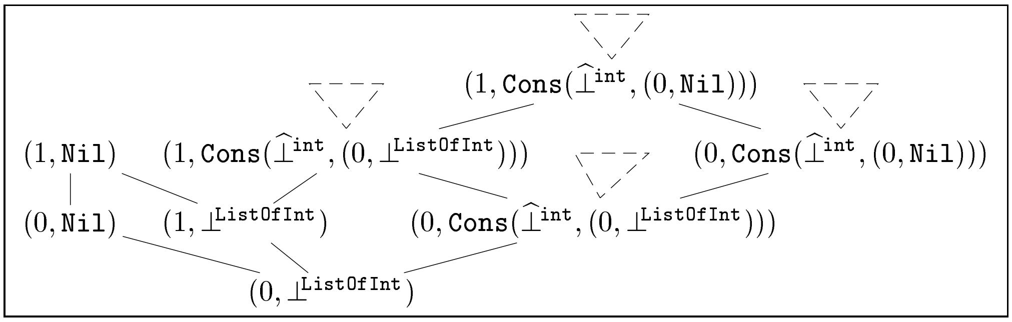
  1. Define f for all $\widehat{\perp}^{t}$.
  2. Define f inductively over t for values not equal to $\widehat{\perp}^{t}$. For $t \in CS$, we define a function f_p with $\widehat{\mathsf{PT}}^{cs}$ as domain first by induction over the term structure and use f_p to define f.

Corollary 3.1
The semantic domains $\widehat{\mathsf{CT}}$ are cpos. $\square$

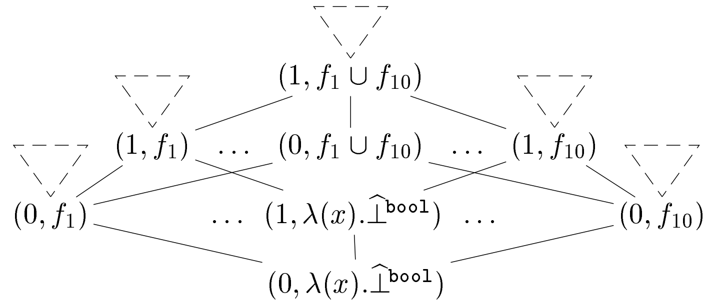
In analogy to Lemma 2.2 the next lemma shows that we can handle $\widehat{\mathsf{CT}}^{cs}$ like a set of terms.



(c) $\widehat{\mathsf{CT}}^{\texttt{bool}}$



(d) $\widehat{\mathsf{PT}}^{\texttt{ListOfInt}}$



where $f_1, \ldots, f_{10} : \widehat{\mathsf{CT}}^{\texttt{bool}} \to \widehat{\mathsf{CT}}^{\texttt{bool}}$ are those monotonic functions which are unequal to $\widehat{\perp}^{\texttt{bool}}$ at one point.

(e) $\widehat{\mathsf{CT}}^{\texttt{bool}\to\texttt{bool}}$

Fig. 3.1: Examples for Augmented Domains

Lemma 3.1 (Unique Decomposition)
Let $\widehat{v} \in \widehat{\mathsf{CT}}^{cs}$ with $\widehat{v} \neq \widehat{\perp}^{cs}$. There exist uniquely determined $n \in \mathbb{N}$, $c \in C^{t_1,\ldots,t_n \to cs}$, $b \in \mathbb{B}$, and $\widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}$ for $1 \leq i \leq n$ such that

$$\widehat{v} = \{(b, c(\widehat{v}'_1, \ldots, \widehat{v}'_n)) \mid \widehat{v}'_i \begin{cases} \in \widehat{v}_i & \text{if } t_i \in CS \\ = \widehat{v}_i & \text{otherwise} \end{cases}, 1 \leq i \leq n\}{\downarrow}$$

We write: $\widehat{v} = (b, c(\widehat{v}_1, \ldots, \widehat{v}_n))$. $\square$

Proof Let $\widehat{v} \in \widehat{\mathsf{CT}}^{cs}$ with $\widehat{v} \neq \widehat{\perp}^{cs} = \{(0, \perp^{cs})\}$. By definition $\widehat{v}$ is an ideal, i.e. $\widehat{v}$ is directed and downward closed: For all $p \in \widehat{v}$ and all $p' \in \widehat{\mathsf{PT}}^{cs}$ with $p' \mathrel{\widehat{\leq}}_{\perp}^{cs} p$ holds that $p \in \widehat{v}$. By definition of $\widehat{\leq}_{\perp}^{cs}$ we can distinguish the following four cases:

1. $p' = (0, \perp^{cs})$
2. $p' = p$
3. $p' = (0, c(\widehat{v}_1, \ldots, \widehat{v}_n))$ and $p = (1, c(\widehat{v}_1, \ldots, \widehat{v}_n))$ with $c \in C^{t_1,\ldots,t_n \to cs}$ and either $\widehat{v}_i \in \widehat{\mathsf{PT}}^{t_i}$ if $t_i \in CS$, or $\widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}$, otherwise $(1 \leq i \leq n)$.
4. $p' = (b, c(\widehat{v}_1, \ldots, \widehat{v}_n))$ and $p = (b, c(\widehat{v}'_1, \ldots, \widehat{v}'_n))$ with $c \in C^{t_1,\ldots,t_n \to cs}$, $b \in \mathbb{B}$, and either $\widehat{v}_i, \widehat{v}'_i \in \widehat{\mathsf{PT}}^{t_i}$ with $\widehat{v}_i \mathrel{\widehat{\leq}}_{\perp}^{t_i} \widehat{v}'_i$ if $t_i \in CS$, or $\widehat{v}_i, \widehat{v}'_i \in \widehat{\mathsf{CT}}^{t_i}$ with $\widehat{v}_i \mathrel{\widehat{\leq}}^{t_i} \widehat{v}'_i$, otherwise.

Hence we have the following representation for all $p \in \widehat{v} \setminus \{(0, \perp^{cs})\}$:

$$p = (b_p, c(v_{p1}, \ldots, v_{pn})) \quad v_{pi} \in \begin{cases} \widehat{\mathsf{PT}}^{t_i} & \text{if } t_i \in CS \\ \widehat{\mathsf{CT}}^{t_i} & \text{otherwise} \end{cases}, 1 \leq i \leq n$$

Now we choose

$$b := \max\{b_p \mid p \in \widehat{v} \setminus \{(0, \perp^{cs})\}\} \qquad \widehat{v}_i := \{v_{pi} \mid p \in \widehat{v} \setminus \{(0, \perp^{cs})\}\} \qquad \text{q.e.d.}$$

Note that in contrast to Lemma 2.2 the downward closure does not only add $(0, \perp^{cs})$ but also values $(0, \ldots)$ if $b = 1$.

With this representation, the augmented values for lists over integers $[n_1, \ldots, n_m]$ can be pictured as shown in Figure 3.2. Here, the tags $b_{s,1}, \ldots, b_{s,m} \in \mathbb{B}$ are used to represent escaping of the *spine* of the list, whereas the tags $b_{e,1}, \ldots, b_{e,m} \in \mathbb{B}$ are added to the *elements* of the list.

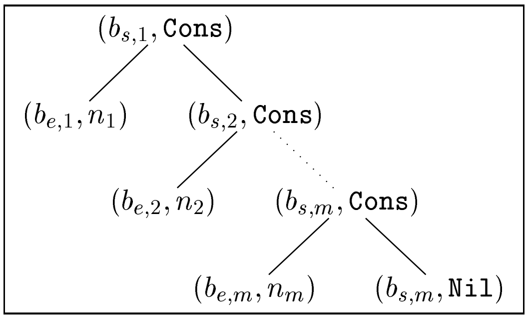


Fig. 3.2: Augmented List of Integers

Augmentation of functional values is more complicated, caused by different kinds of tags. We observe that

- The function is tagged to represent the closure which is created in the execution model for this function.

- The function maps augmented arguments to augmented results, representing the escape behaviour of the function.

For instance, augmented versions of the `append` function can be described by using tags $b_c \in \mathbb{B}$ for the closure, $b_a \in \mathbb{B}$ for the first argument and $b_r \in \mathbb{B}$ for the result:

$$\begin{array}{rcl}
(b_c, \widehat{f}_{\texttt{append}}) & \in & \widehat{\mathsf{CT}}^{\texttt{ListOfInt},\texttt{ListOfInt}\to\texttt{ListOfInt}} \\
\widehat{f}_{\texttt{append}}((b_a, \perp^{\texttt{ListOfInt}}), \widehat{v}_2) & = & (b_r, \perp^{\texttt{ListOfInt}}) \\
\widehat{f}_{\texttt{append}}((b_a, \texttt{Nil}), \widehat{v}_2) & = & \widehat{v}_2 \\
\widehat{f}_{\texttt{append}}((b_a, \texttt{Cons}(a, \widehat{v}_1)), \widehat{v}_2) & = & (b_r, \texttt{Cons}(a, \widehat{f}_{\texttt{append}}(\widehat{v}_1, \widehat{v}_2)))
\end{array}$$

Although our intention was to augment CT, the sets $\widehat{\mathsf{CT}}^{t_1,\ldots,t_n\to t}$ contain more elements than are actually necessary for the augmented semantics. Consider the sets $\widehat{\mathsf{CT}}^{\texttt{bool}\to\texttt{bool}}$. In addition to augmented variants of the functions in $\mathsf{CT}^{\texttt{bool}\to\texttt{bool}}$, this set also contains functions like

$$\begin{array}{rcl}
\widehat{f}_{\text{bad}} : \widehat{\mathsf{CT}}^{\texttt{int}} & \to & \widehat{\mathsf{CT}}^{\texttt{int}} \\
\widehat{f}_{\text{bad}}((b, \perp^{\texttt{int}})) & = & \widehat{\perp}^{\texttt{int}} \\
\widehat{f}_{\text{bad}}((b, v)) & = & (0, b + v)
\end{array}$$

This function obviously has no counterpart without augmentation, because it uses the tag of its argument to obtain the result. However, by definition, such *not well–behaved* functions are never result of the augmented semantics. We formalise this notion in the next section.

3.2 Augmented Denotational Semantics

To reuse as much as possible, we provide a means to convert the semantics of intrinsic functions to the augmented domains. Therefore, we define functions $\oplus$ and $\ominus$, which add a void augmentation and remove augmentation, respectively. Moreover, these functions allow us to formalise the notion of well–behaved functions.

Definition 3.2 ($\ominus$, $\oplus$)
The functions $\ominus : \widehat{\mathsf{CT}} \to \mathsf{CT}$ and $\oplus : \mathsf{CT} \to \widehat{\mathsf{CT}}$ are simultaneously defined as:

1. $\ominus((b, v)) := v$ for all $(b, v) \in \widehat{\mathsf{CT}}^{bs}$,

   $\oplus(v) := (0, v)$ for all $v \in \mathsf{CT}^{bs}$, $bs \in BS$
2. $\ominus(\widehat{v}) := \{\ominus_\perp(p) \mid p \in \widehat{v}\}$ for all $\widehat{v} \in \widehat{\mathsf{CT}}^{cs}$,

   $\oplus(v) := \{\oplus_\perp(p) \mid p \in v\}$ for all $v \in \mathsf{CT}^{cs}$, $cs \in CS$ where the corresponding functions for partial terms $\ominus_\perp(p) : \widehat{\mathsf{PT}}^{cs} \to \mathsf{PT}^{cs}$ and $\oplus_\perp(p) : \mathsf{PT}^{cs} \to \widehat{\mathsf{PT}}^{cs}$ are defined as

   (a) $\ominus_\perp((b, \perp^{cs})) := \perp^{cs}$, $\oplus_\perp(\perp^{cs}) := (0, \perp^{cs})$

(b) $\ominus_\perp((b, c(w_1, \ldots, w_n))) := c(\ominus_1(w_1), \ldots, \ominus_n(w_n))$,
$\oplus_\perp(c(v_1, \ldots, v_n)) := (0, c(\oplus_1(v_1), \ldots, \oplus_n(v_n)))$ for $c \in C^{t_1,\ldots,t_n \to cs}$ and $b \in \mathbb{B}$
with $w_i \in \widehat{\mathsf{PT}}^{t_i}$, $v_i \in \mathsf{PT}^{t_i}$, $\ominus_i = \ominus_\perp$, and $\oplus_i = \oplus_\perp$ if $t_i \in CS$
$w_i \in \widehat{\mathsf{CT}}^{t_i}$, $v_i \in \mathsf{CT}^{t_i}$, $\ominus_i = \ominus$, and $\oplus_i = \oplus$ if $t_i \notin CS$ for $1 \leq i \leq n$.

3. $\ominus((b, f)) := \lambda(v_1, \ldots, v_n). \ominus(f(\oplus(v_1), \ldots, \oplus(v_n)))$ for all $(b, f) \in \widehat{\mathsf{CT}}^{t_1,\ldots,t_n \to t}$.
$\oplus(f) := (0, \lambda(\widehat{v}_1, \ldots, \widehat{v}_n). \oplus(f(\ominus(\widehat{v}_1), \ldots, \ominus(\widehat{v}_n))))$ for all $f \in \mathsf{CT}^{t_1,\ldots,t_n \to t}$. $\triangleleft$

Remarks:

- $\oplus$ always creates augmented values with tags set to zero.
- $\ominus$ simply removes augmentation for basic types.
- For constructed types, we define $\oplus$ and $\ominus$ by using functions $\oplus_\perp$ and $\ominus_\perp$ for partial terms; we ensure that the resulting set is an ideal by considering the down closure. Because $\oplus_\perp$ and $\ominus_\perp$ are monotonic the resulting set is directed.
- For functional types, we are only interested in the result of $\ominus$ for those functions f which do not depend on the augmentation (see remark after Definition 3.1). Therefore, we can define $\ominus$ by using $\oplus$, and hence ignore influences of the augmentation in arguments of f.
- To show that $\oplus$ and $\ominus$ are well–defined, we verify that $\ominus(\widehat{v})$ and $\oplus(v)$ are ideals. Because $\oplus_\perp$ and $\ominus_\perp$ are monotonic $\ominus(\widehat{v})$ and $\oplus(v)$ are directed and it remains to be shown that they are downward closed.
  1. Let $\widehat{v} \in \widehat{\mathsf{CT}}^{cs}$, $p \in \mathsf{PT}^{cs}$, and $p' \in \ominus(\widehat{v})$ such that $p \leq_\perp^{cs} p'$. This means that there is a $\widehat{p}' \in \widehat{v}$ with $p' = \ominus_\perp(\widehat{p}')$. On the other hand it is trivial that $\ominus_\perp$ is surjective and therefore there is also $\widehat{p} \in \widehat{\mathsf{PT}}^{cs}$ in such a way that $p = \ominus_\perp(\widehat{p})$. But since $\ominus_\perp$ is monotonic we also have $\widehat{p} \mathrel{\widehat{\leq}}_\perp^{cs} \widehat{p}'$ which means that $\widehat{p} \in \widehat{v}$ and hence $p = \ominus_\perp(\widehat{p}) \in \ominus(\widehat{v})$.
  2. Let $v \in \mathsf{CT}^{cs}$, $\widehat{p} \in \widehat{\mathsf{PT}}^{cs}$, and $\widehat{p}' \in \oplus_\perp(v)$ such that $\widehat{p} \mathrel{\widehat{\leq}}_\perp^{cs} \widehat{p}'$. This means that there is a $p' \in v$ such that $\widehat{p}' = \oplus_\perp(p')$. Because $\ominus_\perp$ is monotonic and inverse to $\oplus_\perp$ we can conclude that $\ominus_\perp(\widehat{p}) \mathrel{\widehat{\leq}}_\perp^{cs} \ominus_\perp(\widehat{p}') = \ominus_\perp(\oplus_\perp(p')) = p'$. But v is downward closed which means that $\ominus_\perp(\widehat{p}) \in v$ and consequently $\widehat{p} = \oplus(\ominus_\perp(\widehat{p})) \in \oplus(v)$.

Lemma 3.2 (Properties of $\ominus$ and $\oplus$)

1. *$\ominus$ and $\oplus$ preserve types*: for all $t \in \mathsf{T}(S)$

$$\widehat{v} \in \widehat{\mathsf{CT}}^t \iff \ominus(\widehat{v}) \in \mathsf{CT}^t \qquad v \in \mathsf{CT}^t \iff \oplus(v) \in \widehat{\mathsf{CT}}^t$$

2. *$\ominus$ and $\oplus$ are monotonic*: for all $\widehat{v}, \widehat{v}' \in \widehat{\mathsf{CT}}^t$ and $v, v' \in \mathsf{CT}^t$:

$$\widehat{v} \mathrel{\widehat{\leq}}^t \widehat{v}' \implies \ominus(\widehat{v}) \leq^t \ominus(\widehat{v}') \qquad v \leq^t v' \implies \oplus(v) \mathrel{\widehat{\leq}}^t \oplus(v')$$

3. $\ominus$ *and* $\oplus$ *are continuous*: for all directed sets $\widehat{V} \subseteq \widehat{\mathsf{CT}}^t$ and $V \subseteq \mathsf{CT}^t$:
$$\ominus(\widehat{\bigsqcup}^t \widehat{V}) = \bigsqcup^t\{\ominus(\widehat{v}) \mid \widehat{v} \in \widehat{V}\} \qquad \oplus(\bigsqcup^t V) = \widehat{\bigsqcup}^t\{\oplus(v) \mid v \in V\}$$
4. $\ominus$ *is inverse to* $\oplus$: $\ominus \circ \oplus = \mathsf{id}$
5. $\ominus$ *surjective*: $\ominus(\widehat{\mathsf{CT}}) = \mathsf{CT}$
6. $\oplus$ *injective*: for all $v, v' \in \mathsf{CT}^t$ $v \neq v' \implies \oplus(v) \neq \oplus(v')$
7. $\oplus$ *is distributive:* for all $f \in \mathsf{CT}^{t_1,\ldots,t_n \to t}$ and $v_1 \in \mathsf{CT}^{t_1}, \ldots, v_n \in \mathsf{CT}^{t_n}$
$$\oplus(f(v_1, \ldots, v_n)) = (\oplus(f))_2(\oplus(v_1), \ldots, \oplus(v_n))$$ □

PROOF

1. Trivial.
2. Trivial.
3. We give the proof for $\ominus$, the proof for $\oplus$ is can be done analogously. Without loss of generality we can assume that $\widehat{V}$ is not finite.

   Induction on t

   $t = bs \in BS$: All directed sets $\widehat{V} \subseteq \widehat{\mathsf{CT}}^t$ are finite, hence there is nothing to be proved.

   $t = cs \in CS$: By definition we have $\widehat{\bigsqcup}^t \widehat{V} = \bigcup_{\widehat{v} \in \widehat{V}} \widehat{v}$ and hence
$$\begin{aligned}\ominus(\widehat{\bigsqcup}^t \widehat{V}) &= \{\ominus_\perp(p) \mid p \in \widehat{\bigsqcup}^t \widehat{V}\} = \bigcup_{\widehat{v} \in \widehat{V}} \{\ominus_\perp(p) \mid p \in \widehat{v}\} \\ &= \bigcup_{\widehat{v} \in \widehat{V}} \ominus(\widehat{v}) = \bigsqcup^t\{\ominus(\widehat{v}) \mid \widehat{v} \in \widehat{V}\}\end{aligned}$$

   $t = t_1, \ldots, t_m \to t_0$: We have
$$\begin{aligned}\widehat{\bigsqcup}^t \widehat{V} &= (\max\{b \mid (b, f) \in \widehat{V}\},\ \lambda(\widehat{v}_1, \ldots, \widehat{v}_n).\widehat{\bigsqcup}^{t_0}\{f(\widehat{v}_1, \ldots, \widehat{v}_n) \mid (b, f) \in \widehat{V}\}) \\ &=: (b_{\widehat{V}}, f_{\widehat{V}})\end{aligned}$$
   and therefore
$$\begin{aligned}\ominus(\widehat{\bigsqcup}^t \widehat{V}) &= \ominus((b_{\widehat{V}}, f_{\widehat{V}})) = \lambda(v_1, \ldots, v_n).\ominus(f_{\widehat{V}}(\oplus(v_1), \ldots, \oplus(v_n))) \\ &= \lambda(v_1, \ldots, v_n).\ominus(\widehat{\bigsqcup}^{t_0}\{f(\oplus(v_1), \ldots, \oplus(v_n)) \mid (b, f) \in \widehat{V}\} \\ &= \lambda(v_1, \ldots, v_n).\bigsqcup^{t_0}\{\ominus(f(\oplus(v_1), \ldots, \oplus(v_n))) \mid (b, f) \in \widehat{V}\} \\ &= \bigsqcup^t\{\lambda(v_1, \ldots, v_n).\ominus(f(\oplus(v_1), \ldots, \oplus(v_n))) \mid (b, f) \in \widehat{V}\} \\ &= \bigsqcup^t\{\ominus(\widehat{v}) \mid \widehat{v} \in \widehat{V}\}\end{aligned}$$

4. Induction on t

   $t = bs \in BS$: $\ominus(\oplus(v)) = \ominus((0, v)) = v$

   $t = cs \in CS$: It is sufficient to prove that $\ominus_\perp(\oplus_\perp(p)) = p$ for all partial terms $p \in v$ by induction on p:

   $p = \perp^{cs}$: $\ominus_\perp(\oplus_\perp(\perp^{cs})) = \ominus_\perp((0, \perp^{cs})) = \perp^{cs}$

   $$\begin{aligned} p = c(v_1, \dots, v_n)\text{:} \quad & \ominus_\perp(\oplus_\perp(c(v_1, \dots, v_n))) \\ &= \ominus_\perp((0, c(\oplus(v_1), \dots, \oplus(v_n)))) \\ &= c(\ominus(\oplus(v_1)), \dots, \ominus(\oplus(v_n))) \\ &= c(v_1, \dots, v_n) \end{aligned}$$

   $t = t_1, \dots, t_m \to t_0$: Let $\overline{\widehat{v}} := \widehat{v}_1, \dots, \widehat{v}_m$ and $\overline{v} := v_1, \dots, v_m$

   $$\begin{aligned} \ominus(\oplus(v)) &= \ominus((0, \lambda(\overline{\widehat{v}}).\oplus(v(\ominus(\widehat{v}_1), \dots, \ominus(\widehat{v}_m))))) \\ &= \lambda(\overline{v}).\ominus((\lambda(\overline{\widehat{v}}).\oplus(v(\ominus(\widehat{v}_1), \dots, \ominus(\widehat{v}_m))))(\oplus(v_1), \dots, \oplus(v_m))) \\ &= \lambda(\overline{v}).\ominus(\oplus(v(\ominus(\oplus(v_1)), \dots, \ominus(\oplus(v_m))))) \\ &= \lambda(\overline{v}).v(v_1, \dots, v_m) \\ &= v \end{aligned}$$

5. Can directly be concluded from the property that $\ominus$ is inverse to $\oplus$.

6. Trivial.

7. We have $\oplus(f) = (0, \lambda(\widehat{v}_1, \dots, \widehat{v}_n).\oplus(f(\ominus(\widehat{v}_1), \dots, \ominus(\widehat{v}_n))))$. Hence, the right hand side is defined and can be transformed in the following way:

   $$\begin{aligned} & (\oplus(f))_2(\oplus(v_1), \dots, \oplus(v_n)) \\ = \; & \lambda(\widehat{v}_1, \dots, \widehat{v}_n).\oplus(f(\ominus(\widehat{v}_1), \dots, \ominus(\widehat{v}_n))))(\oplus(v_1), \dots, \oplus(v_n)) \\ = \; & \oplus(f(\ominus(\oplus(v_1)), \dots, \ominus(\oplus(v_n)))) \\ = \; & \oplus(f(v_1, \dots, v_n)) \end{aligned}$$

q.e.d.

We use the function $\oplus$ to define the augmented semantics for basic functions.

Definition 3.3 (Augmented Semantics of Intrinsic Functions $\widehat{\mathfrak{M}}$)
Let $f \in \Omega^{t_1, \dots, t_n \to t}$ be an intrinsic function. The *augmented semantics of f* $\widehat{\mathfrak{M}}[\![f]\!] \in \widehat{\mathsf{CT}}^{t_1, \dots, t_n \to t}$ is defined as

1. $\widehat{\mathfrak{M}}[\![bf]\!] := \oplus(\mathfrak{M}[\![bf]\!])$ for all $bf \in BF$

2. $\widehat{\mathfrak{M}}[\![c]\!] := (0, f_c)$ with $f_c = \lambda(\widehat{v}_1, \dots, \widehat{v}_n).(0, c(\widehat{v}_1, \dots, \widehat{v}_n))$

   for all constructors $c \in C$ (see Lemma 3.1)

3. $\widehat{\mathfrak{M}}[\![\mathtt{is}{-}c]\!] := (0, f_{\mathtt{is}-c})$ with $f_{\mathtt{is}-c} = \lambda(\widehat{v}).\begin{cases} (0, \mathtt{true}) & \text{if } \widehat{v} = (b, c(\widehat{v}_1, \dots, \widehat{v}_m)) \\ \widehat{\perp}^{\mathtt{bool}} & \text{if } \widehat{v} = \widehat{\perp}^{cs} \\ (0, \mathtt{false}) & \text{otherwise} \end{cases}$

   for all constructor test $\mathtt{is}{-}c \in CTest$

4. $\widehat{\mathfrak{M}}[\![\mathtt{sel}^j_{}\text{–}c]\!] := (0, f_{\mathtt{sel}^j\text{–}c})$ with $f_{\mathtt{sel}^j\text{–}c} = \lambda(\widehat{v}).\begin{cases} \widehat{v}_j & \text{if } \widehat{v} = (b, c(\widehat{v}_1, \ldots, \widehat{v}_m)),\ 1 \leq j \leq n \\ \widehat{\bot}^t & \text{otherwise} \end{cases}$

   for all selectors $\mathtt{sel}^j\text{–}c \in CSel$ ◁

Examples:

1. Assume that we have a basic function $\mathtt{add} \in BF^{\mathtt{int},\mathtt{int}\to\mathtt{int}}$ interpreted by ϱ as the usual addition on natural numbers, i.e. $\varrho(\mathtt{plus}) = (x, y) \mapsto x + y$. The augmented semantics of `plus` is

$$\begin{aligned} \widehat{\mathfrak{M}}[\![\mathtt{plus}]\!] &= \oplus(\mathfrak{M}[\![\mathtt{plus}]\!]) \\ &= (0, \lambda(\widehat{v}_1, \widehat{v}_2).\, \oplus(\ominus(\widehat{v}_1) + \ominus(\widehat{v}_2))) \\ &= (0, \lambda((b_1, v_1), (b_2, v_2)).(0, v_1 + v_2)) \end{aligned}$$

2. Although we have not yet formally defined how to use such tuples in applications, we can already anticipate that the relation between constructors, test, and selectors is reasonable:

$$\begin{aligned} f_{\mathtt{is}-c'}(f_c(\widehat{v}_1, \ldots, \widehat{v}_n)) &= (0, \mathtt{true}) \iff c = c' \\ f_{\mathtt{sel}^j-c}(f_c(\widehat{v}_1, \ldots, \widehat{v}_n)) &= \widehat{v}_j \text{ for } 1 \leq j \leq n \end{aligned}$$

Note that all escape tags created by the augmented semantics of intrinsic functions are zero. This property is the key feature of the augmented semantics and allows the distinction between values created by the semantics and those values that were stored in environments.

Definition 3.4 (Annotated Operations)
The family of *annotated operations* is defined as $\widehat{\mathsf{Ops}} := \langle \widehat{\mathsf{Ops}}^{t_1,\ldots,t_n\to t} \mid t_1, \ldots, t_n \to t \in \mathsf{T}(S)\rangle$, where $\widehat{\mathsf{Ops}}^{t_1,\ldots,t_n\to t}$ is defined as $\widehat{\mathsf{Ops}}^{t_1,\ldots,t_n\to t} := \widehat{\mathsf{CT}}^{t_1,\ldots,t_n\to t}$. ◁

Definition 3.5 (Augmented Expression Semantics $\widehat{\mathfrak{M}}$)
Let $e \in E^t$ be an expression, $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$ be an environment for variables, and $\widehat{\sigma} \in \mathsf{Env}(DF, \widehat{\mathsf{Ops}})$ be an environment for defined functions. The *augmented semantics of e under $\widehat{\beta}$ and $\widehat{\sigma}$* ($\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma}) \in \widehat{\mathsf{CT}}^t$) is inductively defined:

- $\widehat{\mathfrak{M}}[\![x]\!](\widehat{\beta}, \widehat{\sigma}) := \widehat{\beta}(x)$ for $x \in X^t$
- $\widehat{\mathfrak{M}}[\![F]\!](\widehat{\beta}, \widehat{\sigma}) := \widehat{\sigma}(F)$ for $F \in DF^t$
- $\widehat{\mathfrak{M}}[\![f]\!](\widehat{\beta}, \widehat{\sigma}) := \widehat{\mathfrak{M}}[\![f]\!]$ for $f \in \Omega^t$
- $\widehat{\mathfrak{M}}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\widehat{\beta}, \widehat{\sigma}) := \begin{cases} f(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \ldots, \widehat{\mathfrak{M}}[\![e_n]\!](\widehat{\beta}, \widehat{\sigma})) & \text{if } m = n \\ (b, f') & \text{if } 1 \leq m < n \end{cases}$

  with $\begin{aligned} (b, f) &= \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta}, \widehat{\sigma}), \\ f' &= \lambda(\widehat{v}_{m+1}, \ldots, \widehat{v}_n).f(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \ldots, \widehat{\mathfrak{M}}[\![e_m]\!](\widehat{\beta}, \widehat{\sigma}), \widehat{v}_{m+1}, \ldots, \widehat{v}_n) \end{aligned}$

  for $e_0 \in E^{t_1,\ldots,t_n\to t_0}$ and $e_i \in E^{t_i}$ $(1 \leq i \leq m)$

- $\widehat{\mathfrak{M}}[\![\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2]\!](\widehat{\beta},\widehat{\sigma}) := \begin{cases} \widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta},\widehat{\sigma}) & \text{if } \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta},\widehat{\sigma}) = (b,\texttt{true}), \\ \widehat{\bot}^t & \text{if } \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta},\widehat{\sigma}) = \widehat{\bot}^{\texttt{bool}} \\ \widehat{\mathfrak{M}}[\![e_2]\!](\widehat{\beta},\widehat{\sigma}) & \text{if } \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta},\widehat{\sigma}) = (b,\texttt{false}) \end{cases}$ for $e_0 \in E^{\texttt{bool}}$ and $e_1, e_2 \in E^t$ ◁

If an application is saturated we drop the escape tag and apply the actual function. For a partial application we propagate the escape tag. This reflects the operational behaviour that a partial application is represented by a newly created closure containing all arguments of the partial application.

Again, all escape tags created by the augmented semantics of expressions are zero.

Example: For the right hand side

$$\begin{aligned} e = {} & \texttt{if (is-Nil l) then l'} \\ & \texttt{else (Cons (sel}^1\texttt{-Cons l) (append (sel}^2\texttt{-Cons l) l'))} \end{aligned}$$

of the `append` definition, a variable environment $\widehat{\beta} = [\texttt{l}/(1,\texttt{Nil}), \texttt{l'}/\widehat{v}']$, and any environment $\widehat{\sigma}$ for defined functions, we get

$$\begin{aligned} \widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta},\widehat{\sigma}) &= \begin{cases} \widehat{\mathfrak{M}}[\![\texttt{l'}]\!](\widehat{\beta},\widehat{\sigma}) & \text{if } \widehat{v}_1 = (b,\texttt{true}) \\ \widehat{\bot}^t & \text{if } \widehat{v}_1 = \widehat{\bot}^{\texttt{bool}} \\ \widehat{\mathfrak{M}}[\![\texttt{(Cons ...)}]\!](\widehat{\beta},\widehat{\sigma}) & \text{if } \widehat{v}_1 = (b,\texttt{false}) \end{cases} \\ & \quad \text{where } \widehat{v}_1 = \widehat{\mathfrak{M}}[\![\texttt{(is-Nil l)}]\!](\widehat{\beta},\widehat{\sigma}) \\ & \qquad\qquad\quad = f_{\texttt{is-Nil}}(\widehat{\mathfrak{M}}[\![\texttt{l}]\!](\widehat{\beta},\widehat{\sigma})) \\ & \qquad\qquad\quad = (0,\texttt{true}) \\ &= \widehat{\mathfrak{M}}[\![\texttt{l'}]\!](\widehat{\beta},\widehat{\sigma}) = \widehat{\beta}(\texttt{l'}) = \widehat{v}' \end{aligned}$$ ◇

Definition 3.6 (Augmented Program Semantics $\widehat{\mathfrak{M}}$)
Given a program $P = (F_j(x_{j1},\ldots,x_{jn_j}) := e_j \mid 1 \le j \le p)$ with $F_j \in DF^{t_{j1},\ldots,t_{jn_j} \to t_j}$, $x_{ji} \in X^{t_{ji}}$, and $e_j \in E^{t_j}$ with variables $\{x_{j1},\ldots,x_{jn_j}\}$ for $1 \le j \le p$, the *augmented semantics of* P is an environment $\widehat{\mathfrak{M}}[\![P]\!] \in \mathsf{Env}(DF, \widehat{\mathsf{CT}})$ defined as

$$\widehat{\mathfrak{M}}[\![P]\!] := [F_1/(0, \mathsf{fix}(\Phi_{\widehat{\mathfrak{M}},P})_1), \ldots, F_p/(0, \mathsf{fix}(\Phi_{\widehat{\mathfrak{M}},P})_p)]$$

Here, $\mathsf{fix}(\Phi_{\widehat{\mathfrak{M}},P})$ is the least fixpoint of the transformation $\Phi_{\widehat{\mathfrak{M}},P} : \mathsf{FS}_{\widehat{\mathfrak{M}},P} \to \mathsf{FS}_{\widehat{\mathfrak{M}},P}$ on the function space

$$\mathsf{FS}_{\widehat{\mathfrak{M}},P} := \prod_{j=1}^{p} \left[\widehat{\mathsf{CT}}^{t_{j1}} \times \cdots \times \widehat{\mathsf{CT}}^{t_{jn_j}} \to \widehat{\mathsf{CT}}^{t_j} \right]$$

The transformation is defined as

$$\Phi_{\widehat{\mathfrak{M}},P}(g_1,\ldots,g_n) := \begin{pmatrix} \lambda(v_{11},\ldots,v_{1n_1}).\widehat{\mathfrak{M}}[\![e_1]\!]([x_{11}/v_{11},\ldots,x_{1n_1}/v_{1n_1}],\widehat{\sigma}) \\ \vdots \\ \lambda(v_{p1},\ldots,v_{pn_p}).\widehat{\mathfrak{M}}[\![e_p]\!]([x_{p1}/v_{p1},\ldots,x_{pn_p}/v_{pn_p}],\widehat{\sigma}) \end{pmatrix}$$
$$\text{where } \widehat{\sigma} := [F_1/(0,g_1),\ldots,F_p/(0,g_p)]$$ ◁

Because the semantic domains are cpo's and the expression semantics defines continuous functions, the fixpoint theorem of Knaster and Tarski (Theorem A.1) guarantees that $\mathsf{fix}(\Phi_{\widehat{\mathfrak{M}},P})$ exists and can be represented in the following way:

$$\mathsf{fix}(\Phi_{\widehat{\mathfrak{M}},P}) = \bigsqcup\{\Phi^i_{\widehat{\mathfrak{M}},P}(\bot^{\mathsf{FS}_{\widehat{\mathfrak{M}},P}}) \mid i \in \mathbb{N}\}$$

Hence, $\widehat{\mathfrak{M}}[\![P]\!]$ is well–defined.

Example: For the `append` program $P_{\texttt{append}}$, we have the transformation

$$\Phi_{\widehat{\mathfrak{M}},P_{\texttt{append}}}(\widehat{g}) = \lambda(\widehat{v}_1,\widehat{v}_2).\widehat{\mathfrak{M}}[\![\texttt{if (is-Nil l) ...}]\!]([\texttt{l}/\widehat{v}_1,\texttt{l'}/\widehat{v}_2],[\texttt{append}/\widehat{g}])$$

Successive application of $\Phi_{\widehat{\mathfrak{M}},P_{\texttt{append}}}$ on the least element $\bot = \lambda(\widehat{x},\widehat{y}).\widehat{\bot}^{\texttt{ListOfInt}}$ of the associated function space $\mathsf{FS}_{\widehat{\mathfrak{M}},P_{\texttt{append}}} = \widehat{\mathsf{CT}}^{\texttt{ListOfInt},\texttt{ListOfInt}\to\texttt{ListOfInt}}$ yields the following sequence:

$$\begin{array}{rcl}
\Phi^1_{\widehat{\mathfrak{M}},P_{\texttt{append}}}(\bot) & = & (\widehat{v}_1,\widehat{v}_2) \mapsto \begin{cases} \widehat{v}_2 & \text{if } \widehat{v}_1 = (b,\texttt{Nil}) \\ \widehat{\bot}^{\texttt{ListOfInt}} & \text{otherwise} \end{cases} \\
\Phi^2_{\widehat{\mathfrak{M}},P_{\texttt{append}}}(\bot) & = & (\widehat{v}_1,\widehat{v}_2) \mapsto \begin{cases} \widehat{v}_2 & \text{if } \widehat{v}_1 = (b,\texttt{Nil}) \\ (0,\texttt{Cons}(n,\widehat{v}_2)) & \text{if } \widehat{v}_1 = (b_1,\texttt{Cons}(n,(b_2,\texttt{Nil}))) \\ (0,\texttt{Cons}(n,\widehat{\bot}^{\texttt{ListOfInt}})) & \\ & \text{if } \widehat{v}_1 = (b_1,\texttt{Cons}(n,(b_2,\bot^{\texttt{ListOfInt}}))) \\ \widehat{\bot}^{\texttt{ListOfInt}} & \text{otherwise} \end{cases} \\
& \vdots & \\
\Phi^m_{\widehat{\mathfrak{M}},P_{\texttt{append}}}(\bot) & = & (\widehat{v}_1,\widehat{v}_2) \mapsto \begin{cases} \widehat{v}_2 & \text{if } \widehat{v}_1 = (b,\texttt{Nil}) \\ (0,\texttt{Cons}(n,\widehat{v}_2)) & \text{if } \widehat{v}_1 = (b_1,\texttt{Cons}(n,(b_2,\texttt{Nil}))) \\ (0,\texttt{Cons}(n,\widehat{\bot}^{\texttt{ListOfInt}})) & \\ & \text{if } \widehat{v}_1 = (b_1,\texttt{Cons}(n,(b_2,\bot^{\texttt{ListOfInt}}))) \\ \vdots & \\ (0,\texttt{Cons}(n_1,\ldots(0,\texttt{Cons}(n_m,\widehat{v}_2))\ldots)) & \\ & \text{if } \widehat{v}_1 = (b_1,\texttt{Cons}(n_1,\ldots(b_{m+1},\texttt{Nil})\ldots)) \\ (0,\texttt{Cons}(n_1,\ldots(0,\texttt{Cons}(n_m,\widehat{\bot}))\ldots)) & \\ & \text{if } \widehat{v}_1 = (b_1,\texttt{Cons}(n_1,\ldots(b_{m+1},\bot)\ldots)) \\ \widehat{\bot}^{\texttt{ListOfInt}} & \text{otherwise} \end{cases}
\end{array}$$

$\diamond$

3.3 Augmentation is Well–Behaved

Besides augmented variants of standard values, the sets $\widehat{\mathsf{CT}}^{t_1,\ldots,t_n\to t}$ contain much more than actually necessary for the augmented semantics. In this section we introduce the notion of well–behaved values and show that the augmented semantics only creates well–behaved values.

We use the dual property to Lemma 3.2 (Item 7) as a characterisation of well–behaved functions.

DEFINITION 3.7 (WELL–BEHAVED FUNCTION/VALUE/ENVIRONMENT)

- $\widehat{v} \in \widehat{\mathsf{CT}}^S$ is *well–behaved.*
- $(b, f) \in \widehat{\mathsf{CT}}^{t_1,\ldots,t_n \to t}$ is *well–behaved* iff we have for all well–behaved arguments $\widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}$

$$\ominus(f(\widehat{v}_1, \ldots, \widehat{v}_n)) = (\ominus((b, f)))(\ominus(\widehat{v}_1), \ldots, \ominus(\widehat{v}_n))$$

  and $f(\widehat{v}_1, \ldots, \widehat{v}_n)$ is *well–behaved.*
- $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$ is *well–behaved* iff $\widehat{\beta}(x)$ is well–behaved for all $x \in X$. $\lhd$

Intuitively, a function is well–behaved if it does not use the augmentation of its arguments for the computation of its result. We check this relationship by giving an alternative characterisation:

$$\begin{aligned} & \ominus(f(\widehat{v}_1, \ldots, \widehat{v}_n)) = (\ominus((b, f)))(\ominus(\widehat{v}_1), \ldots, \ominus(\widehat{v}_n)) \\ \iff \quad & \ominus(f(\widehat{v}_1, \ldots, \widehat{v}_n)) = \ominus(f(\oplus(\ominus(\widehat{v}_1)), \ldots, \oplus(\ominus(\widehat{v}_n)))) \end{aligned}$$

Example: The counterexample $\widehat{f}_{\mathrm{bad}}$ from the last section is *not* well–behaved:

$$\begin{aligned} \ominus(\widehat{f}_{\mathrm{bad}}(\oplus(\ominus((1,0))))) &= \ominus(\widehat{f}_{\mathrm{bad}}((0,0))) = \ominus((0,0)) = 0 \\ &\neq 1 = \ominus(0,1) = \ominus(\widehat{f}_{\mathrm{bad}}((1,0))) \end{aligned} \quad \diamond$$

The following lemma justifies that this definition is reasonable wrt. the unaugmented values.

LEMMA 3.3
Let $v \in \mathsf{CT}^t$ be an unaugmented value. Then $\oplus(v)$ is well–behaved. $\Box$

PROOF Induction on t

$t = s \in S$: Trivial.

$t = t_1, \ldots, t_m \to t_0$: $\quad \ominus(((\oplus(f))_2)(\widehat{v}_1, \ldots, \widehat{v}_n))$

$$\begin{aligned} &= \ominus(\lambda(\widehat{v}_1, \ldots, \widehat{v}_n).\, \oplus(f(\ominus(\widehat{v}_1), \ldots, \ominus(\widehat{v}_n)))(\widehat{v}_1, \ldots, \widehat{v}_n)) \\ &= \ominus(\oplus(f(\ominus(\widehat{v}_1), \ldots, \ominus(\widehat{v}_n)))) \\ &= f(\ominus(\widehat{v}_1), \ldots, \ominus(\widehat{v}_n)) \\ &= (\ominus(\oplus(f)))(\ominus(\widehat{v}_1), \ldots, \ominus(\widehat{v}_n)) \end{aligned}$$

Because $((\oplus(f))_2)(\widehat{v}_1, \ldots, \widehat{v}_n) = \oplus(f(\ominus(\widehat{v}_1), \ldots, \ominus(\widehat{v}_n)))$ it is well–behaved by induction hypothesis. q.e.d.

We verify that not well–behaved functions are never result of the augmented semantics.

LEMMA 3.4
If $e \in E^t$ is an expression and $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$, $\widehat{\sigma} \in \mathsf{Env}(DF, \widehat{\mathsf{Ops}})$ are well–behaved environments, then we have that $\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma})$ is well–behaved. $\Box$

PROOF Induction on e

$e = x \in X$: $\widehat{\beta}$ well–behaved.

$e = F \in DF$: $\widehat{\sigma}$ well–behaved.

$e = f \in \Omega$: Trivial.

$e = (e_0\ e_1\ \dots\ e_m)$: The induction hypothesis is that $\widehat{\mathfrak{M}}[\![e_i]\!](\widehat{\beta}, \widehat{\sigma})$ is well–behaved. Let $f = (\widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta}, \widehat{\sigma}))_2$. We can observe two cases:

1. $m = n$: $\widehat{\mathfrak{M}}[\![(e_0\ e_1\ \dots\ e_n)]\!](\widehat{\beta}, \widehat{\sigma}) = f(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \dots, \widehat{\mathfrak{M}}[\![e_n]\!](\widehat{\beta}, \widehat{\sigma}))$ is well–behaved because f is well–behaved and application of a well–behaved function to well-behaved arguments yields a well–behaved result.
2. $m < n$: Let $\widehat{v}_{m+1} \in \widehat{\mathsf{CT}}^{t_{m+1}}, \dots, \widehat{v}_n \in \widehat{\mathsf{CT}}^{t_n}$:
$$\begin{aligned}&\ominus((\widehat{\mathfrak{M}}[\![(e_0\ e_1\ \dots\ e_m)]\!](\widehat{\beta}, \widehat{\sigma}))_2(\widehat{v}_{m+1}, \dots, \widehat{v}_n))\\ &= \ominus(f(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \dots, \widehat{\mathfrak{M}}[\![e_m]\!](\widehat{\beta}, \widehat{\sigma}), \widehat{v}_{m+1}, \dots, \widehat{v}_n))\\ &= \ominus(f(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \dots, \widehat{\mathfrak{M}}[\![e_m]\!](\widehat{\beta}, \widehat{\sigma}), \oplus(\ominus(\widehat{v}_{m+1})), \dots, \oplus(\ominus(\widehat{v}_n))))\\ &= (\ominus(\widehat{\mathfrak{M}}[\![(e_0\ e_1\ \dots\ e_m)]\!](\widehat{\beta}, \widehat{\sigma})))(\ominus(\widehat{v}_{m+1}), \dots, \ominus(\widehat{v}_n))\end{aligned}$$
and the application is well–behaved because application of well–behaved functions to well–behaved arguments yield well–behaved results.

$e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: Trivial. q.e.d.

To transfer this result to the semantics of programs, we have to ensure that *abstraction* of well–behaved values yields a well–behaved value.

Lemma 3.5
Let $e \in E^t$ be an expression with variables $x_1, \dots, x_n$ and $\widehat{\sigma} \in \mathsf{Env}(DF, \widehat{\mathsf{Ops}})$ be a well–behaved environment. Then $(0, \lambda(\widehat{v}_1, \dots, \widehat{v}_n).\widehat{\mathfrak{M}}[\![e]\!]([x_1/\widehat{v}_1, \dots, x_n/\widehat{v}_n], \widehat{\sigma}))$ well–behaved. □

Proof Let $\widehat{v}_1, \dots, \widehat{v}_m \in \widehat{\mathsf{CT}}$ well–behaved, $f = \lambda(\widehat{v}_1, \dots, \widehat{v}_m).\widehat{\mathfrak{M}}[\![e]\!]([x_1/\widehat{v}_1, \dots, x_m/\widehat{v}_m], \widehat{\sigma})$. With Lemma 3.4 we know that

$$f(\widehat{v}_1, \dots, \widehat{v}_m) = \widehat{\mathfrak{M}}[\![e]\!]([x_1/\widehat{v}_1, \dots, x_m/\widehat{v}_m], \widehat{\sigma})$$

is well–behaved. It remains to show that

$$\begin{aligned}&\ominus(f(\widehat{v}_1, \dots, \widehat{v}_m)) = (\ominus((b, f)))(\ominus(\widehat{v}_1), \dots, \ominus(\widehat{v}_m))\\ \iff &\ominus(\widehat{\mathfrak{M}}[\![e]\!]([x_1/\widehat{v}_1, \dots, x_m/\widehat{v}_m], \widehat{\sigma})) = \ominus(\widehat{\mathfrak{M}}[\![e]\!]([x_1/\oplus(\ominus(\widehat{v}_1)), \dots, x_m/\oplus(\ominus(\widehat{v}_m))], \widehat{\sigma}))\end{aligned}$$

We proof the latter by induction on e:

$e = x \in X$:
$$\begin{aligned}&\ominus(\widehat{\mathfrak{M}}[\![e]\!]([x_1/\widehat{v}_1, \dots, x_m/\widehat{v}_m], \widehat{\sigma})) = \ominus(\widehat{v}_i)\\ =\ &\ominus(\oplus(\ominus(\widehat{v}_i))) = \ominus(\widehat{\mathfrak{M}}[\![e]\!]([x_1/\oplus(\ominus(\widehat{v}_1)), \dots, x_m/\oplus(\ominus(\widehat{v}_m))], \widehat{\sigma}))\end{aligned}$$

$e = F \in DF$: Trivial.

$e = f \in \Omega$: Trivial.

$e = (e_0\ e_1\ \ldots\ e_m)$: The induction hypothesis (IH) is that:

$$\ominus(\widehat{\mathfrak{M}}[\![e_i]\!]([x_1/\widehat{v}_1,\ldots,x_m/\widehat{v}_m],\widehat{\sigma})) = \ominus(\widehat{\mathfrak{M}}[\![e_i]\!]([x_1/\oplus(\ominus(\widehat{v}_1)),\ldots,x_m/\oplus(\ominus(\widehat{v}_m))],\widehat{\sigma}))$$

$$\begin{array}{lrcl} \text{Let} & (b,f) & = & \widehat{\mathfrak{M}}[\![e_0]\!]([x_1/\widehat{v}_1,\ldots,x_m/\widehat{v}_m],\widehat{\sigma}), \\ & (b',f') & = & \widehat{\mathfrak{M}}[\![e_0]\!]([x_1/\oplus(\ominus(\widehat{v}_1)),\ldots,x_m/\oplus(\ominus(\widehat{v}_m))],\widehat{\sigma}) \end{array}$$

Two cases may occur:

1. $n = m$:
$$\begin{array}{l} \ominus(\widehat{\mathfrak{M}}[\![e]\!]([x_1/\widehat{v}_1,\ldots,x_m/\widehat{v}_m],\widehat{\sigma})) \\ = \ominus(f(\widehat{\mathfrak{M}}[\![e_1]\!]([x_1/\widehat{v}_1,\ldots,x_m/\widehat{v}_m],\widehat{\sigma}),\ldots,\widehat{\mathfrak{M}}[\![e_n]\!]([x_1/\widehat{v}_1,\ldots,x_m/\widehat{v}_m],\widehat{\sigma}))) \\ \overset{(Lemma\ 3.4)}{=} \ominus((b,f))(\ominus(\widehat{\mathfrak{M}}[\![e_1]\!]([x_1/\widehat{v}_1,\ldots],\widehat{\sigma})),\ldots,\ominus(\widehat{\mathfrak{M}}[\![e_n]\!]([x_1/\widehat{v}_1,\ldots],\widehat{\sigma})))) \\ \overset{(IH)}{=} \ominus((b',f'))(\ominus(\widehat{\mathfrak{M}}[\![e_1]\!]([x_1/\oplus(\ominus(\widehat{v}_1)),\ldots],\widehat{\sigma})),\ldots) \\ \overset{(Lemma\ 3.4)}{=} \ominus(f'(\ominus(\widehat{\mathfrak{M}}[\![e_1]\!]([x_1/\oplus(\ominus(\widehat{v}_1)),\ldots],\widehat{\sigma})),\ldots)) \\ = \ominus(\widehat{\mathfrak{M}}[\![e]\!]([x_1/\oplus(\ominus(\widehat{v}_1)),\ldots,x_m/\oplus(\ominus(\widehat{v}_m))],\widehat{\sigma})) \end{array}$$
2. $n < m$: Analogously.

$e =$ **if** e_0 **then** e_1 **else** e_2: Trivial. q.e.d.

Corollary 3.2 ($\widehat{\mathfrak{M}}$ is Well–Behaved)
For all programs P holds that $\widehat{\mathfrak{M}}[\![P]\!]$ is well–behaved. □

Proof Immediately follows from the last theorem and Item 3 of Lemma 3.2. q.e.d.

3.4 Augmentation is a Conservative Extension

We verify that the augmentation does not change the computed value, i.e. that the augmented semantics $\widehat{\mathfrak{M}}$ is a *conservative extension* of $\mathfrak{M}$. We formalise the notion of conservative extension in the following way.

Definition 3.8 (Conservative Extension)
Let $\widehat{v} \in \widehat{\mathsf{CT}}^t$ be an augmented value and $v \in \mathsf{CT}^t$ be a value. We say that $\widehat{v}$ *is a conservative extension* of v iff $\ominus(\widehat{v}) = v$. ◁

Lemma 3.6
Let $f \in \Omega^{t_1,\ldots,t_n \to t}$ be an intrinsic function. $\widehat{\mathfrak{M}}[\![f]\!]$ is a conservative extension of $\mathfrak{M}[\![f]\!]$. □

Proof We distinguish four cases:

1. $f = bf \in BF$ $\mathfrak{M}[\![f]\!] = \ominus(\oplus(\mathfrak{M}[\![f]\!])) = \ominus(\widehat{\mathfrak{M}}[\![f]\!])$
2. $f = c \in C$
$$\begin{array}{l} \mathfrak{M}[\![c]\!] = \lambda(v_1,\ldots,v_n).c(v_1,\ldots,v_n) \\ = \lambda(v_1,\ldots,v_n).\{c(v_1',\ldots,v_n') \mid v_i' \begin{cases} \in v_i & \text{if } t_i \in CS \\ = v_i & \text{otherwise} \end{cases}, 1 \le i \le n\}{\downarrow} \end{array}$$

$$
\begin{aligned}
&= \lambda(v_1,\ldots,v_n).\{c(v'_1,\ldots,v'_n) \mid v'_i \begin{cases} \in \ominus(\oplus(v_i)) & \text{if } t_i \in CS \\ = \ominus(\oplus(v_i)) & \text{otherwise} \end{cases}, 1 \le i \le n\}{\downarrow} \\
&= \lambda(v_1,\ldots,v_n).\{c(\ominus_1(\widehat{v}'_1),\ldots,\ominus_n(\widehat{v}'_n)) \mid \begin{cases} \ominus_i = \ominus_\perp, \widehat{v}'_i \in \oplus(v_i) & t_i \in CS \\ \ominus_i = \ominus, \widehat{v}'_i = \oplus(v_i) & \text{o/w} \end{cases}, 1 \le i \le n\}{\downarrow} \\
&= \lambda(v_1,\ldots,v_n).\{\ominus_\perp((b, c(\widehat{v}'_1,\ldots,\widehat{v}'_n))) \mid \widehat{v}'_i \begin{cases} \in \oplus(v_i) & t_i \in CS \\ = \oplus(v_i) & \text{o/w} \end{cases}, 1 \le i \le n\}{\downarrow} \\
&= \lambda(v_1,\ldots,v_n).\ominus(\{(b, c(\widehat{v}'_1,\ldots,\widehat{v}'_n)) \mid \widehat{v}'_i \begin{cases} \in \oplus(v_i) & t_i \in CS \\ = \oplus(v_i) & \text{o/w} \end{cases}, 1 \le i \le n\}{\downarrow}) \\
&= \lambda(v_1,\ldots,v_n).\ominus(c(\oplus(v_1),\ldots,\oplus(v_n))) \\
&= \ominus((0, \lambda(\widehat{v}_1,\ldots,\widehat{v}_n).(0, c(\widehat{v}_1,\ldots,\widehat{v}_n)))) \\
&= \ominus(\widehat{\mathfrak{M}}[\![c]\!])
\end{aligned}
$$

3. $f = \mathtt{is}{-}c \in CTest$

$$
\begin{aligned}
\mathfrak{M}[\![\mathtt{is}{-}c]\!] &= \lambda(v).\begin{cases} \mathtt{true} & \text{if } v = c(v_1,\ldots,v_m) \\ \perp^{\mathtt{bool}} & \text{if } v = \perp^{cs} \\ \mathtt{false} & \text{otherwise} \end{cases} \\
&= \lambda(v).\begin{cases} \ominus((0,\mathtt{true})) & \text{if } \oplus(v) = (b, c(\widehat{v}_1,\ldots,\widehat{v}_m)) \\ \ominus(\widehat{\perp}^{\mathtt{bool}}) & \text{if } \oplus(v) = \widehat{\perp}^{cs} \\ \ominus((0,\mathtt{false})) & \text{otherwise} \end{cases} \\
&= \ominus((0, \lambda(\widehat{v}).\begin{cases} (0,\mathtt{true}) & \text{if } \widehat{v} = (b, c(\widehat{v}_1,\ldots,\widehat{v}_m)) \\ \widehat{\perp}^{\mathtt{bool}} & \text{if } \widehat{v} = \widehat{\perp}^{cs} \\ (0,\mathtt{false}) & \text{otherwise} \end{cases})) \\
&= \ominus(\widehat{\mathfrak{M}}[\![\mathtt{is}{-}c]\!])
\end{aligned}
$$

4. $f = \mathtt{sel}^{j}{-}c \in CSel$

$$
\begin{aligned}
\mathfrak{M}[\![\mathtt{sel}^{j}{-}c]\!] &= \lambda(v).\begin{cases} v_j & \text{if } v = c(v_1,\ldots,v_m),\ 1 \le j \le m \\ \perp^t & \text{otherwise} \end{cases} \\
&= \lambda(v).\begin{cases} v_j & \text{if } v = c(\ominus(\oplus(v_1)),\ldots,\ominus(\oplus(v_m))),\ 1 \le j \le m \\ \perp^t & \text{otherwise} \end{cases} \\
&= \lambda(v).\begin{cases} \ominus(\widehat{v}_j) & \text{if } \oplus(v) = (b, c(\oplus(v_1),\ldots,\oplus(v_m))),\ 1 \le j \le m \\ \ominus(\widehat{\perp}^t) & \text{otherwise} \end{cases} \\
&= \lambda(v).\begin{cases} \ominus(\widehat{v}_j) & \text{if } \oplus(v) = (b, c(\widehat{v}_1,\ldots,\widehat{v}_m)),\ 1 \le j \le m \\ \ominus(\widehat{\perp}^t) & \text{otherwise} \end{cases} \\
&= \ominus((0, \lambda(\widehat{v}).\begin{cases} \widehat{v}_j & \text{if } \widehat{v} = (b, c(\widehat{v}_1,\ldots,\widehat{v}_m)),\ 1 \le j \le m \\ \widehat{\perp}^t & \text{otherwise} \end{cases})) \\
&= \ominus(\widehat{\mathfrak{M}}[\![\mathtt{sel}^{j}{-}c]\!])
\end{aligned}
$$

q.e.d.

To use $\ominus$ to relate the original and the modified semantics, we extend the functions $\ominus$ and $\oplus$, and the notion of conservative extension to environments.

Definition 3.9 ($\ominus$, $\oplus$, and Conservative Extension for Environments)
Let $X = \langle X^t \mid t \in \mathsf{T}(S)\rangle$ be a family of variables and $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$, $\beta \in \mathsf{Env}(X, \mathsf{CT})$ environments. We define $\ominus(\widehat{\beta}) \in \mathsf{Env}(X, \mathsf{CT})$ and $\oplus(\beta) \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$:

$$
\begin{aligned}
(\ominus(\widehat{\beta}))(x) &:= \ominus(\widehat{\beta}(x)) && \forall x \in X \\
(\oplus(\beta))(x) &:= \oplus(\beta(x)) && \forall x \in X
\end{aligned}
$$

Furthermore, $\widehat{\beta}$ is a *conservative extension* of β iff $\widehat{\beta}(x)$ is a conservative extension of $\beta(x)$ for all $x \in X$. $\triangleleft$

THEOREM 3.1
Let $e \in E^t$ be an expression and $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$, $\widehat{\sigma} \in \mathsf{Env}(DF, \widehat{\mathsf{Ops}})$ be well–behaved environments. $\widehat{\mathfrak{M}}$ is a conservative extension of $\mathfrak{M}$, i.e. the following diagram commutes

$$\begin{tikzcd}
(\widehat{\beta}, \widehat{\sigma}) \arrow[r, "\widehat{\mathfrak{M}}[\![e]\!]"] \arrow[d, "\ominus"'] & \widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma}) \arrow[d, "\ominus"] \\
(\ominus(\widehat{\beta}), \ominus(\widehat{\sigma})) \arrow[r, "\mathfrak{M}[\![e]\!]"'] & \mathfrak{M}[\![e]\!](\ominus(\widehat{\beta}), \ominus(\widehat{\sigma}))
\end{tikzcd}$$ $\square$

PROOF Induction on e

$e = x \in X$: $\mathfrak{M}[\![x]\!](\ominus(\widehat{\beta}), \ominus(\widehat{\sigma})) = \ominus(\widehat{\beta})(x) = \ominus(\widehat{\beta}(x)) = \ominus(\widehat{\mathfrak{M}}[\![x]\!](\widehat{\beta}, \widehat{\sigma}))$

$e = F \in DF$: Analogous to last case.

$e = f \in \Omega$: Lemma 3.6

$e = (e_0\ e_1\ \dots\ e_m)$:

Induction hypothesis: $\mathfrak{M}[\![e_i]\!](\ominus(\widehat{\beta}), \ominus(\widehat{\sigma})) = \ominus(\widehat{\mathfrak{M}}[\![e_i]\!](\widehat{\beta}, \widehat{\sigma}))$ for $0 \le i \le n$.
Let $f = \mathfrak{M}[\![e_0]\!](\ominus(\widehat{\beta}), \ominus(\widehat{\sigma}))$ and $\widehat{f} = (\widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta}, \widehat{\sigma}))_2$.
Then we have for all well–behaved $\widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}$ $(1 \le i \le n)$:

$$\begin{aligned}
f(\ominus(\widehat{v_1}), \dots, \ominus(\widehat{v_n})) &= (\mathfrak{M}[\![e_0]\!](\ominus(\widehat{\beta}), \ominus(\widehat{\sigma})))(\ominus(\widehat{v_1}), \dots, \ominus(\widehat{v_n})) \\
&= \ominus(\widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta}, \widehat{\sigma}))(\ominus(\widehat{v_1}), \dots, \ominus(\widehat{v_n})) \\
&\overset{(*)}{=} \ominus((\widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta}, \widehat{\sigma}))_2(\widehat{v}_1, \dots, \widehat{v}_n)) \\
&= \ominus(\widehat{f}(\widehat{v}_1, \dots, \widehat{v}_n))
\end{aligned}$$

The equation $(*)$ holds because with Lemma 3.3 we know that $\widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta}, \widehat{\sigma})$ is well–behaved. We distinguish two cases:

1. $m = n$: $\begin{aligned}[t] (\mathfrak{M}[\![e]\!](\ominus(\widehat{\beta}), \ominus(\widehat{\sigma}))) &= f(\mathfrak{M}[\![e_1]\!](\ominus(\widehat{\beta}), \ominus(\widehat{\sigma})), \dots, \mathfrak{M}[\![e_n]\!](\ominus(\widehat{\beta}), \ominus(\widehat{\sigma}))) \\ &= f(\ominus(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma})), \dots, \ominus(\widehat{\mathfrak{M}}[\![e_n]\!](\widehat{\beta}, \widehat{\sigma}))) \\ &= \ominus(\widehat{f}(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \dots, \widehat{\mathfrak{M}}[\![e_n]\!](\widehat{\beta}, \widehat{\sigma}))) \\ &= \ominus(\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma})) \end{aligned}$
2. $m < n$: analogously

$e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: Trivial. q.e.d.

This theorem directly leads to the main result of this section, which shows that the augmented semantics preserves the original computations.

COROLLARY 3.3 ($\widehat{\mathfrak{M}}$ IS CONSERVATIVE EXTENSION OF $\mathfrak{M}$)
For all programs P we have that $\widehat{\mathfrak{M}}[\![P]\!]$ is a conservative extension of $\mathfrak{M}[\![P]\!]$. $\square$

PROOF Immediately follows from the last theorem and Item 3 of Lemma 3.2. q.e.d.

3.5 Escaping as (Augmented) Denotational Property

We use the augmentation to define our notion of escaping. We distinguish between 'original' and 'copy' by adding non–zero tags to the original; a 'copy' created by the augmented semantics has only tags which are equal to zero: Its augmentation is 'void' (or 'blank').

DEFINITION 3.10 (VOID AUGMENTATION)
We define that a well–behaved $\widehat{v} \in \widehat{\mathsf{CT}}^t$ has a *void augmentation* by induction on t:

$t = s \in S$: $\exists v \in \mathsf{CT}^t : \oplus(v) = \widehat{v}$

$t = t_1, \ldots, t_m \to t_0$: $\widehat{v} = (0, f)$ such that $f(\widehat{v}_1, \ldots, \widehat{v}_n)$ has void augmentation for all arguments $\widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}$ $(1 \leq i \leq n)$ with void augmentation.

Furthermore, an environment $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$ has a *void augmentation* iff $\widehat{\beta}(a)$ has a void augmentation for all $a \in X$. ◁

For basic sorts and constructed sorts the definition is straightforward. For functional values, we only require that the function result has void augmentation for arguments which also have this property. Otherwise, already the identity like (b, id) would never have void–augmentation.

Obviously, trivially augmented values have void augmentation.

LEMMA 3.7
For all $v \in \mathsf{CT}^t$: $\oplus(v)$ has void augmentation. □

PROOF Induction on t

$t = s \in S$: trivial

$t = t_1, \ldots, t_m \to t_0$: Let $\widehat{v}_1 \in \widehat{\mathsf{CT}}^{t_1}, \ldots, \widehat{v}_m \in \widehat{\mathsf{CT}}^{t_m}$ with void augmentation.

$$((\oplus(f))_2)(\widehat{v}_1, \ldots, \widehat{v}_n) = \oplus(v(\ominus(\widehat{v}_1), \ldots, \ominus(\widehat{v}_n)))) \in \widehat{\mathsf{CT}}^{t_0}$$

has void augmentation by induction hypothesis. q.e.d.

Figure 3.3 shows the relation between augmented values, well–behaved values, values with void augmentation, and embedded unaugmented values.

The next lemma shows that non–zero tags are not created spontaneously by the augmented semantics.

LEMMA 3.8
Let $e \in E^t$ be an expression and $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$, $\widehat{\sigma} \in \mathsf{Env}(DF, \widehat{\mathsf{Ops}})$ be environments with void augmentation. Then $\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma})$ has void augmentation. □

PROOF Induction on e

$e = x \in X$: $\widehat{\beta}$ has void augmentation.

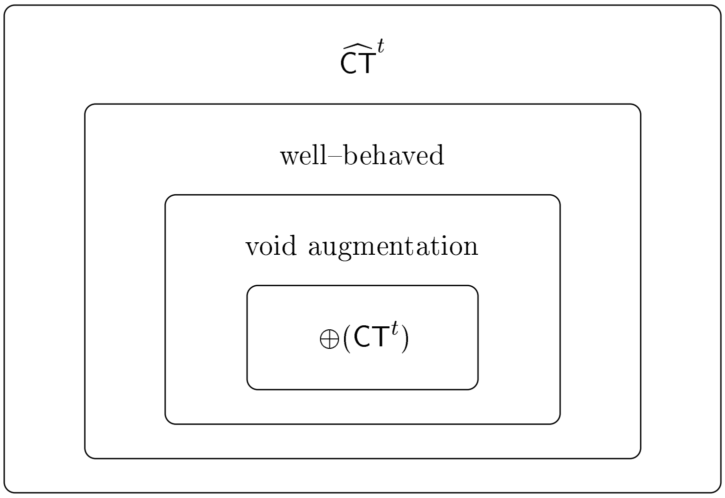


Fig. 3.3: Relation Augmented/Well–behaved/Void Augmented/Unaugmented Values

$e = F \in DF$: $\widehat{\sigma}$ has void augmentation.

$e = f \in \Omega$: Obvious.

$e = (e_0\ e_1\ \ldots\ e_m)$: By induction hypothesis we know that $\widehat{\mathfrak{M}}[\![e_i]\!](\widehat{\beta}, \widehat{\sigma})$ has void augmentation. Let $(b, f) = \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta}, \widehat{\sigma})$. With Lemma 3.4 we have that $\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma})$ is well–behaved. We distinguish two cases:

1. $m = n$: $\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma}) = f(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \ldots, \widehat{\mathfrak{M}}[\![e_n]\!](\widehat{\beta}, \widehat{\sigma}))$ has void augmentation by definition.
2. $m < n$: We have $b = 0$ by induction hypothesis. Furthermore, let $\widehat{v}_j \in \widehat{\mathsf{CT}}^{t_j}$ with void augmentation for $m + 1 \leq j \leq n$. By definition

$$f'(\widehat{v}_{m+1}, \ldots, \widehat{v}_n) = f(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \ldots, \widehat{\mathfrak{M}}[\![e_m]\!](\widehat{\beta}, \widehat{\sigma}), \widehat{v}_{m+1}, \ldots, \widehat{v}_n)$$

has void augmentation and hence $\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma}) = (b, f')$ has void augmentation.

$e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: Trivial. q.e.d.

The major point in the definition of escaping is the property that values with non–void augmentation cause the result to have non–void augmentation. Since we know from the last lemma that the augmented semantics does not create non–void tags by itself, we can be sure that this approach is reasonable.

Definition 3.11 (Escaping)

1. Let $\widehat{f} : \widehat{\mathsf{CT}}^{t_1} \times \cdots \times \widehat{\mathsf{CT}}^{t_n} \to \widehat{\mathsf{CT}}^{t}$ be a function and let $\widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}$ $(1 \leq i \leq n)$ be values such that only $\widehat{v}_j$ has non–void augmentation. We define that *parts of* $\widehat{v}_j$ *escape from the application* $\widehat{f}(\widehat{v}_1, \ldots, \widehat{v}_n)$ iff $\widehat{f}(\widehat{v}_1, \ldots, \widehat{v}_n)$ has non–void augmentation.

2. Let $e \in E^t$ be an expression, $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$, $\widehat{\sigma} \in \mathsf{Env}(DF, \widehat{\mathsf{Ops}})$ be environments, and $x \in X$ be a variable such that $\widehat{\beta}(y)$ has void augmentation iff $x \neq y$. We define that *parts of x escape from e under $\widehat{\sigma}$* iff $\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma})$ has non–void augmentation.

3. Let P be a program, $e \in E^t$ be an expression, and $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$ be an environment. We define that *parts of x escape from e* iff parts of x escape from e under $\widehat{\mathfrak{M}}[\![P]\!]$. $\lhd$

Examples:

1. Consider the `append` program from the beginning of this chapter in conjunction with the expression `(append l1 l2)`. Furthermore, we define environments

   - $\widehat{\beta}_{s,x} := [x/(1, \texttt{Cons}((0, 42), (1, \texttt{Nil}))]$ augmenting the *spine* of the list, i.e. the constructors.
   - $\widehat{\beta}_{e,x} := [x/(0, \texttt{Cons}((1, 42), (0, \texttt{Nil}))]$ augmenting the entry 42.
   - $\widehat{\beta}_{t,x} := [x/(0, \texttt{Cons}((0, 42), (0, \texttt{Nil}))]$ augmenting nothing.

   for all $x \in X$. We now can observe four significant situations:

   (a) $\widehat{\mathfrak{M}}[\![\texttt{(append l1 l2)}]\!](\widehat{\beta}_{t,\texttt{l1}} \cup \widehat{\beta}_{e,\texttt{l2}}, \mathfrak{M}[\![P]\!])$
   $= (0, \texttt{Cons}((0, 42), (0, \texttt{Cons}((1, 42), (0, \texttt{Nil})))))$
   The entry of the second list escapes from the expression.

   (b) $\widehat{\mathfrak{M}}[\![\texttt{(append l1 l2)}]\!](\widehat{\beta}_{t,\texttt{l1}} \cup \widehat{\beta}_{s,\texttt{l2}}, \mathfrak{M}[\![P]\!])$
   $= (0, \texttt{Cons}((0, 42), (1, \texttt{Cons}((0, 42), (1, \texttt{Nil})))))$
   The spine of the second list escapes.

   (c) $\widehat{\mathfrak{M}}[\![\texttt{(append l1 l2)}]\!](\widehat{\beta}_{e,\texttt{l1}} \cup \widehat{\beta}_{t,\texttt{l2}}, \mathfrak{M}[\![P]\!])$
   $= (0, \texttt{Cons}((1, 42), (0, \texttt{Cons}((0, 42), (0, \texttt{Nil})))))$
   The entry of the first list escapes.

   (d) $\widehat{\mathfrak{M}}[\![\texttt{(append l1 l2)}]\!](\widehat{\beta}_{s,\texttt{l1}} \cup \widehat{\beta}_{t,\texttt{l2}}, \mathfrak{M}[\![P]\!])$
   $= (0, \texttt{Cons}((0, 42), (0, \texttt{Cons}((0, 42), (0, \texttt{Nil})))))$
   This is a result with void augmentation, i.e. the spine of the first list does *not escape* from the expression. This is because `append` copies the spine of the first list.

2. As an example for higher–order escaping, we consider the `filter` program:

```
filter p []    = []
filter p (a:l) = if (p a) then (a:(filter p l)) else (filter p l)
```

   Here, we want to investigate if `p` escapes[1]. Therefore, we consider the expression $e =$ `filter p [1,2,3]` and the environment $\widehat{\beta} := [\texttt{p}/(1, \lambda(x). \begin{cases} (0, \texttt{true}) & \text{if } x = 2 \\ (0, \texttt{false}) & \text{otherwise} \end{cases})]$. Evaluating e with this environment yields

$$\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \mathfrak{M}[\![P]\!]) = (0, \texttt{Cons}((0, 2), (0, \texttt{Nil})))$$

[1] In this case, it is already obvious from the type of `filter` (`(a->Bool) -> [a] -> [a]`) that its first argument cannot escape.

which has void augmentation.

3. A simple higher–order example where escaping occurs is `apply`, defined as `apply f x = (f x)`. We consider the expression e = `apply f 0`, where `f` is a variable of type $\mathtt{int}, \mathtt{int} \to \mathtt{int}$ and the environment $\widehat{\beta} := [\mathtt{f}/(1, f)]$. Then we have

$$\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \mathfrak{M}[\![P]\!]) = (1, \lambda(\widehat{y}).(f((0,0), \widehat{y})))$$

which has a non–void augmentation.

4. Of course, escaping not only depends on the *shape* of the data structures, but also on the *content*. Again, we consider the `filter` program, but now with expression e = `filter p l` and the environments $\widehat{\beta}_{\mathtt{false}} := [\mathtt{p}/(0, \lambda(x).(0, \mathtt{false})), \mathtt{l}/(1, \mathtt{Nil})]$ and $\widehat{\beta}_{\mathtt{true}} := [\mathtt{p}/(0, \lambda(x).(0, \mathtt{true})), \mathtt{l}/(1, \mathtt{Nil})]$ Evaluating e with these environments yields the expected results

$$\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}_{\mathtt{false}}, \mathfrak{M}[\![P]\!]) = (0, \mathtt{Nil}) \qquad \widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}_{\mathtt{true}}, \mathfrak{M}[\![P]\!]) = (1, \mathtt{Nil})$$

The abstract interpretation presented in the next chapter approximates escaping by analysing the shape of the data only. Therefore, the abstract interpretation fails to find non–escaping in this case.

3.6 Summary

In this chapter, we have formalised the notion of escaping. Since escaping cannot be expressed with the standard semantics $\mathfrak{M}$, we have introduced the semantics $\widehat{\mathfrak{M}}$, a conservative extension of $\mathfrak{M}$. This semantics uses augmented domains to allow the formalisation of escaping. The augmented domains have been defined by tagging the standard domains with binary values, such that zero tags are always created by the semantics $\widehat{\mathfrak{M}}$. Non–zero tags can be used to identify values not created by $\widehat{\mathfrak{M}}$.

4. The Abstract Interpretation

All abstract interpretations essentially consist of two abstraction steps:

1. *abstraction of domains*
2. *abstraction of terms*

Since we have to guarantee the termination of the abstract interpretation, the domains we choose must have finite ascending chains only. Therefore it is vital to find a way to handle recursive data structures.

4.1 Abstract Domains

To obtain abstract domains from the concrete domains $\widehat{\mathsf{CT}}^t$ we conceptually perform two steps. The first step is to remove the basic values from the "leafs" of the term and to keep only the structure and the escape tags. For the moment, we call these *intermediate sets* I^t. In Figure 4.1 we demonstrate this step for $t = \texttt{ListOfInt}$. Like the original domains, the elements of I^t have no bound and hence the sets I^t are infinite.

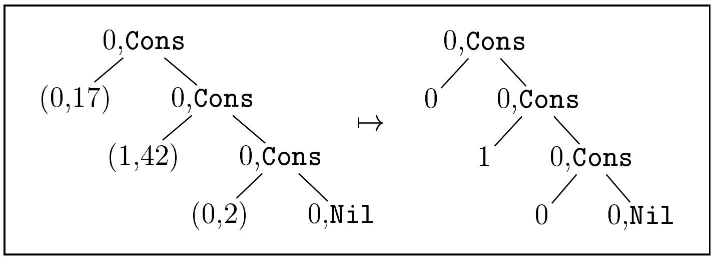


Fig. 4.1: $\widehat{\mathsf{CT}}^{\texttt{ListOfInt}}$ to $I^{\texttt{ListOfInt}}$

The second step on the way to the abstract domains is based on the observation that the escape tags in $i \in I^t$ can be grouped in *levels* corresponding to the structure of i. For instance, integers are used to build lists of integers, which in turn are used to build lists of lists of integers. In this case we have three levels: the escape tags of integer entries, of constructors of integer lists, and of constructors of top–level lists. If I^t and hence CT^t has n levels, we define the corresponding abstract domains to be isomorphic to $\mathbb{B}^n$. Each component of the abstract value is obtained as the *maximum of all entries of level* i (see Figure 4.2). The underlying abstraction is the assumption that all elements of the same level behave in the same way wrt. escaping.[1]

[1] There is an obvious simplification of the domains, which could be applied here. Instead of $\mathbb{B}^n$ we could

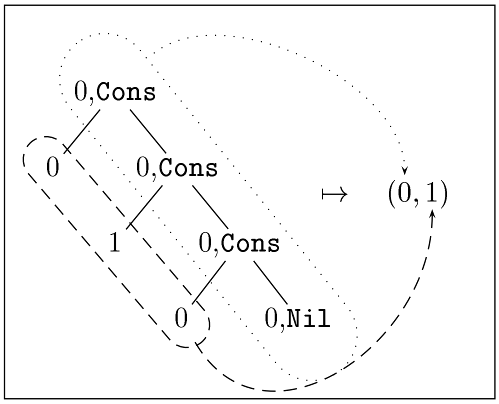


Fig. 4.2: $I^{\texttt{ListOfInt}}$ to $A^{\texttt{ListOfInt}}$

Our first attempt to define the abstract sets starts with setting $A^{bs} := \mathbb{B}$ for all basic sorts $bs \in BS$. We then can define the sets A^{cs} for the constructed sorts as the least solution of the equation system

$$A^{cs} = \mathbb{B} \times \prod_{c \in C^{t_1,\ldots,t_n \to cs}} \prod_{\substack{1 \le i \le n \\ t_i \ne cs}} A^{t_i} \qquad \forall cs \in CS$$

The idea is as follows: For each constructor, we store a tuple of the abstract values for the "non–recursive parameters". In addition, we add one tag for the behaviour of the whole structure and all of its recursive substructures.

To evaluate the first Cartesian product $\prod$ we need an order on all constructors of target sort cs. The second product filters out all direct recursiveness since we want to combine all constructors of cs in one level. If we apply this to our example, we can compute

$$\begin{aligned} A^{\texttt{ListOfInt}} &= \mathbb{B} \times \prod_{c \in \{\texttt{Nil}^{\to \texttt{ListOfInt}}, \texttt{Cons}^{\texttt{int},\texttt{ListOfInt} \to \texttt{ListOfInt}}\}} \prod_{\substack{1 \le i \le n \\ t_i \ne \texttt{ListOfInt}}} A^{t_i} = \mathbb{B} \times A^{\texttt{int}} \\ &= \mathbb{B}^2 \end{aligned}$$

Note that we assume that a product ranging over an empty index creates the neutral element for subsequent products. Therefore, these parts do not contribute to the resulting domain. E.g. the constructor `Nil` has no influence.

But there is a fatal error in this approach: It is possible that we have a constructed sort cs, with $|A^{cs}| = \infty$. A minimal example for this consists of the data definitions:

```
datatype T1 ::= C1 T2
datatype T2 ::= C2 T1
```

choose $\{0, \ldots, n\}$, with a value k representing that levels k and above do not escape. This approach is taken in [Hug92] and [PG92]. The underlying observation is that it is not possible to define a functional program where a certain constructor c escapes and simultaneously a constructor d below c does not. However, the drawback is that this approach works for list structures, but *not* for all data structures.

which are valid definitions in Miranda or Haskell, and which can be expressed in our abstract syntax by choosing $C^{T2 \to T1} = \{\texttt{C1}\}$ and $C^{T1 \to T2} = \{\texttt{C2}\}$. The corresponding semantic domains have exactly one element:

$$\begin{aligned} \mathsf{CT}^{T1} &= \{\texttt{C1(C2(C1(}\cdots\texttt{)))}\} \\ \mathsf{CT}^{T2} &= \{\texttt{C2(C1(C2(}\cdots\texttt{)))}\} \end{aligned}$$

Accordingly, we get the abstract set equations:

$$\begin{aligned} A^{\texttt{T1}} &= \mathbb{B} \times A^{\texttt{T2}} \\ A^{\texttt{T2}} &= \mathbb{B} \times A^{\texttt{T1}} \end{aligned}$$

which have the unique solution $A^{\texttt{T1}} = A^{\texttt{T2}} = \mathbb{B}^\omega$, the set of all infinite sequences of booleans. Of course, sets of this kind are not useful for abstract interpretation, because to guarantee termination of the abstract interpretation we need sets with only finite ascending chains.

Therefore, we must enhance our notion. Our first attempt suffered from the different management of *direct recursion* and *indirect recursion*. In the above example, it would be convenient to choose $A^{\texttt{T1}} = A^{\texttt{T2}} = \mathbb{B}^1$: One level for all occurrences of `C1` and `C2`.

To consider a more intricate example, we use the following types:

```
datatype ITree ::= ILeaf int  | INode int  CTree CTree
datatype CTree ::= CLeaf char | CNode char ITree ITree
```

They define types of trees consisting of alternating layers of `int` resp. `char` entries. Obviously, we have five levels:

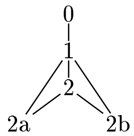
- Two levels for the `int`–entries in the constructors `ILeaf` and `INode`.
- Two for the `char`–entries in `CLeaf` and `CNode`.
- One level for all constructors[2].

Hence, we have $A^{\texttt{ITree}} = A^{\texttt{CTree}} = \mathbb{B} \times (\mathbb{B}^2)^2$.

To formalise the notion of indirect recursion between constructed sorts, we introduce the notion of *dependence*.

[2] The indirect recursion, or, to be more precise, the fact that the constructed sorts need not form a proper hierarchy is also the reason for the failure of the simplified approach sketched above. It implicitly assumes that the abstract levels of the type can be ordered in a single chain, which is not always possible. The three levels cannot be ordered linearly, since the levels for the `int` and for the `char` entries are incomparable.

An extension of the simplified approach would be to choose a domain where several incomparable levels are positioned as successors of a common predecessor. Since all incomparable levels can be handled independently we have to add values for all combinations of escape of levels. In our case, this would be the domain to the left where the '2' represents a situation where both `char` and `int` entries escape but not the constructors. These domains, however, have lost the simplicity of a linear chain, so there is no advantage anymore.



DEFINITION 4.1 (DEPENDENCE)
Let $cs_1, cs_2 \in CS$. We say that cs_1 *depends on* cs_2 ($cs_1 \twoheadleftarrow cs_2$) iff there exists a constructor $c \in C^{t_1,\ldots,t_{i-1},cs_2,t_{i+1},\ldots,t_n \to cs_1}$. As usual, $\overset{\star}{\twoheadleftarrow}$ denotes the transitive and reflexive closure of $\twoheadleftarrow$. If we have $cs_1 \overset{\star}{\twoheadleftarrow} cs_2$ and $cs_2 \overset{\star}{\twoheadleftarrow} cs_1$, then we say that cs_1 and cs_2 are *mutually recursive dependent*. By definition, this is an equivalence relation, and we denote the equivalence class of $cs \in CS$ by

$$[cs] := \{cs' \in CS \mid cs' \overset{\star}{\twoheadleftarrow} cs \text{ and } cs \overset{\star}{\twoheadleftarrow} cs'\}$$

We extent the notion of equivalence to all sorts by defining $[bs] := \{bs\}$ for all basic sorts. $\lhd$

Intuitively, constructed sorts are ordered in a *dependence graph* $(CS, \twoheadleftarrow)$ such that the equivalence classes are the strongly connected components. Switching to the *factor graph* $(CS/\overset{\star}{\twoheadleftarrow}, \twoheadleftarrow / \overset{\star}{\twoheadleftarrow})$ obviously yields a *hierarchy*. Therefore, we can associate with each $cs \in CS$ a height in this hierarchy:

DEFINITION 4.2 (HEIGHT $\mathsf{h}(cs)$)
Let $cs \in CS$ be a constructed sort. The *height of cs* $\mathsf{h}(cs) \in \mathbb{N}$ is defined as

$$\mathsf{h}(cs) := 1 + \max_{cs' \in CS,\ [cs'] \neq [cs],\ cs \twoheadleftarrow cs'} \mathsf{h}(cs') \qquad \lhd$$

With these preparations, we can define the abstract domains:

DEFINITION 4.3 (ABSTRACT DOMAINS)
For all $t \in \mathsf{T}(S)$ we define the *abstract domain* $\langle \mathsf{A}^t, \preceq^t \rangle$

1. $\mathsf{A}^{bs} = \mathbb{B}$ and $\perp^{bs} := 0 \preceq^{bs} 1$ for all basic sort $bs \in BS$.

2. For all constructed sorts $cs \in CS$, we define A^{cs} as the (unique) solution of the following equations:

$$\mathsf{A}^{cs} := \mathbb{B} \times \prod_{cs' \in [cs]} \prod_{c \in C^{t_1,\ldots,t_n \to cs'}} \prod_{[t_i] \neq [cs]} \mathsf{A}^{t_i} \qquad (*)$$

   Furthermore, we define $\preceq^{cs}$ component–wise.

3. $\mathsf{A}^{t_1,\ldots,t_n \to t} := \langle \mathbb{B}, \leq \rangle \times [\langle \mathsf{A}^{t_1}, \preceq^{t_1} \rangle \times \cdots \times \langle \mathsf{A}^{t_n}, \preceq^{t_n} \rangle \to \langle \mathsf{A}^t, \preceq^t \rangle]$ for all functional types $t_1, \ldots, t_n \to t \in \mathsf{T}(S)$. $\lhd$

Remarks:

- The first component of each abstract value is the binary tag indicating escape wrt. the corresponding level. The domain for a constructed sort cs is obtained by gluing the domains of all non recursive (within $[cs]$) arguments of all constructors of a constructed sort cs' in the class $[cs]$.

- In Chapter 2, we introduced F with the restriction, that constructors must have first–order type. If we drop this restriction, the domains A^t would be not well–defined! For instance, a definition like

```
datatype T ::= c (T -> T)
```

would result in the abstract domain

$$\mathsf{A}^{\mathtt{T}} = \mathbb{B} \times \mathsf{A}^{\mathtt{T}\to\mathtt{T}} = \mathbb{B} \times (\mathbb{B} \times [\mathsf{A}^{\mathtt{T}} \to \mathsf{A}^{\mathtt{T}}])$$

A solution to this equation would be an infinite domain, which is not useful as abstract domain. See Chapter 8 for further details on this topic.

LEMMA 4.1
The abstract domains $\langle \mathsf{A}^t, \preceq^t \rangle$ are well–defined for all $t \in \mathsf{T}(S)$. □

PROOF All we have to prove is that the equations $(*)$ have an unique solution for all $cs \in CS$. Therefore, it suffices to prove that for all $cs' \in [cs]$ and $c \in C^{t_1,\ldots,t_n \to cs'}$ holds that $t_i \in CS$ implies $\mathsf{h}(t_i) < \mathsf{h}(cs)$. With this result, the solution of $(*)$ can be computed deterministically by induction on $\mathsf{h}(cs)$.

Let cs', c, and t_i fulfill the above conditions.

$$\begin{array}{rcl} \mathsf{h}(cs) & = & 1 + \max\limits_{cs'' \in CS,\ [cs''] \neq [cs],\ cs \twoheadleftarrow cs''} \mathsf{h}(cs'') \\ & > & \max\limits_{cs'' \in CS,\ [cs''] \neq [cs],\ cs \twoheadleftarrow cs''} \mathsf{h}(cs'') \\ & \geq & \mathsf{h}(cs'') \quad \forall cs'' \in CS,\ [cs''] \neq [cs],\ cs \twoheadleftarrow cs'' \\ & \geq & \mathsf{h}(cs'') \quad \forall cs'' \in CS,\ [cs''] \neq [cs],\ \exists c \in C^{t_1,\ldots,t_{i-1},cs'',t_{i+1},\ldots,t_n \to cs} \end{array}$$ q.e.d.

LEMMA 4.2
For all $t \in \mathsf{T}(S)$ exists $n_t \in \mathbb{N}$ such that $\langle \mathsf{A}^t, \preceq^t \rangle \simeq \langle \mathbb{B}^{n_t}, \leq_{n_t} \rangle \simeq \langle \mathfrak{P}(\{1, \ldots, 2^{n_t}\}), \subseteq \rangle$ where $\leq_{n_t}$ is the "bitwise less or equal" defined as

$$\begin{array}{rcl} \leq_n : \mathbb{B}^n \times \mathbb{B}^n & \to & \mathbb{B} \\ (a_1, \ldots, a_n) \leq_n (b_1, \ldots, b_n) & = & a_1 \leq b_1 \wedge \ldots \wedge a_n \leq b_n \end{array}$$ □

PROOF Induction on t

$t = bs \in BS$: Choose $n_t = 1$.

$t = cs \in CS$: For all $cs \in CS$, we can find n_{cs} as the unique solution of the following equations:

$$n_{cs} = 1 + \prod_{cs' \in [cs]} \prod_{c \in C^{t_1,\ldots,t_n \to cs'}} \prod_{[t_i] \neq [cs]} n_{t_i} \qquad (**)$$

The proof that the equations $(**)$ have an unique solution is analogous to the proof of Lemma 4.1.

$t = t_1, \ldots, t_m \to t_0$: By induction hypothesis we have natural numbers $n_0, \ldots, n_m$ such that

$$\langle \mathsf{A}^{t_i}, \preceq^{t_i} \rangle \simeq \langle \mathbb{B}^{n_{t_i}}, \leq_{n_{t_i}} \rangle$$

Each function in $[\mathsf{A}^{t_1} \times \cdots \times \mathsf{A}^{t_m} \to \mathsf{A}^{t_0}]$ can be represented by its (finite) graph. Since the domain is finite ($2^{n_{t_1} + \cdots + n_{t_m}}$ elements), we can fix an enumeration of all elements of the domain:

$$\mathsf{A}^{t_1} \times \cdots \times \mathsf{A}^{t_m} = \{e_1, \ldots, e_{2^{n_{t_1} + \cdots + n_{t_m}}}\}$$

Now we can represent the graph as a $2^{n_{t_1}+\cdots+n_{t_m}}$–tuple of function results:

$$(f(e_1), \ldots, f(e_{2^{n_{t_1}+\cdots+n_{t_m}}})) \in (\mathsf{A}^{t_0})^{2^{n_{t_1}+\cdots+n_{t_m}}}$$

In this representation, $f_1(a_1, \ldots, a_m) \preceq^t f_2(a_1, \ldots, a_m)$ for all $a_i \in \mathsf{A}^{t_i}$ is equivalent to

$$(f_1(e_1), \ldots, f_1(e_{2^{n_{t_1}+\cdots+n_{t_m}}})) \leq_{2^{n_{t_0}} \cdot 2^{n_{t_1}+\cdots+n_{t_m}}} (f_2(e_1), \ldots, f_2(e_{2^{n_{t_1}+\cdots+n_{t_m}}}))$$

Obviously, we have $n_t = 1 + 2^{n_{t_0}+n_{t_1}+\cdots+n_{t_m}}$, i.e.

$$\langle \mathsf{A}^{t_1,\ldots,t_m \to t_0},\ \preceq^{t_1,\ldots,t_m \to t_0} \rangle \simeq \langle \mathbb{B}^{1+2^{n_{t_0}+n_{t_1}+\cdots+n_{t_m}}},\ \leq_{1+2^{n_{t_0}+n_{t_1}+\cdots+n_{t_m}}} \rangle \qquad \text{q.e.d.}$$

In Section 4.5 we show how we can reduce the exponential factor in the representation of function types to a linear factor. The underlying idea is instead of representing functions by their full graph, we use only the function values for certain *test arguments*. This can occur, because we can reconstruct the complete graph of functions created during the abstract interpretation solely by their values for the test arguments. This modified representation allows an efficient implementation of the abstract interpretation.

COROLLARY 4.1 (PROPERTIES OF $\langle \mathsf{A}^t,\ \preceq^t \rangle$)

- $\langle \mathsf{A}^t,\ \preceq^t \rangle$ is finite
- $\langle \mathsf{A}^t,\ \preceq^t \rangle$ is a complete lattice $\langle \mathsf{A}^t;\ \sqcup^t,\ \sqcap^t \rangle$ such that $\sqcup^t \simeq \vee_{n_t}$ and $\sqcap^t \simeq \wedge_{n_t}$. □

Remarks:

- The finiteness guarantees termination of the abstract interpretation.
- The representation of $\sqcup^t$ as "bitwise or" and $\sqcap^t$ as "bitwise and" is again an indication that an implementation can be very efficient: Both operations are very cheap.

4.2 Interpretation of Selectors and Constructors

To formalise which parts of an abstract value are affected by a *particular constructor (or selector)* of a *particular type* of an equivalence class $[cs]$, we need arbitrary, but fixed orderings on all constructed types of class $[cs]$ and on all constructors of a target type $cs' \in [cs]$. More formally, we assume that for all $cs \in CS$

1. $[cs] = \{cs_1, \ldots, cs_{n_{cs}}\}$
2. $C^{cs} = \{c_1, \ldots, c_{m_{cs}}\}$

Implicitly, we have already used these orderings in the definition of A^{cs}.

DEFINITION 4.4 (RECURSIVE/NON–RECURSIVE ARGUMENTS)
Let $c \in C^{t_1,\ldots,t_n \to cs}$ be a constructor. For all $1 \leq i \leq n$

- t_i is called a *recursive argument of* c iff $[t_i] = [cs]$

- t_i is called a *non–recursive argument of* c iff $[t_i] \neq [cs]$ $\lhd$

Without loss of generality, we assume that all constructors have their recursive arguments after their non–recursive arguments, i.e. for all $c \in C^{t_1,\ldots,t_n \to cs}$ exists $1 \leq r_c \leq n+1$ such that

$$t_i \notin [cs] \text{ for } 1 \leq i < r_c \qquad \text{and} \qquad t_j \in [cs] \text{ for } r_c \leq j \leq n+1$$

Of course, this is no real restriction, because we can always reorder argument sequences. However, without this restriction the formalisation would become more difficult, because the abstract interpretation of selectors and constructors handles recursive and non–recursive arguments differently [Moh95a, Moh95b].

Examples:

1. For the list constructors, we have $r_{\texttt{Nil}} = 1$ and $r_{\texttt{Cons}} = 2$.
2. For the `CTree`/`ITree` constructors, we have $r_{\texttt{ILeaf}} = r_{\texttt{CLeaf}} = 1$ and $r_{\texttt{INode}} = r_{\texttt{CNode}} = 2$.

With this assumption, we can reformulate equations $(*)$ in Definition 4.3:

Corollary 4.2 (Alternative Characterisation of A^{cs})

$$\mathsf{A}^{cs} = \mathbb{B} \times \prod_{cs' \in [cs]} \; \prod_{c \in C^{t_1,\ldots,t_n \to cs'}} \; \prod_{i=1}^{r_c - 1} \mathsf{A}^{t_i} \qquad \square$$

Definition 4.5 (Escape Semantics of Constructors)
Let $c \in C^{t_1,\ldots,t_n \to cs}$ be a constructor such that cs has index l in the order of the sorts and c has index m in the order of constructors. The *escape semantics of* c is defined as

$$\begin{array}{rcl} \mathfrak{E} & : & C^{t_1,\ldots,t_n \to cs} \to \mathsf{A}^{t_1,\ldots,t_n \to cs} \\ \mathfrak{E}[\![c]\!] & := & (0, \lambda(a_1,\ldots,a_n).(0, \bot_{<l}, (\bot_{l,<m}, (a_1,\ldots,a_{r_c-1}), \bot_{l,>m}), \bot_{>l}) \sqcup \bigsqcup_{i=r_c}^{n} a_i) \end{array}$$

where $\bot_{<l}$ is the least element in $\prod_{k=1}^{l-1} \prod_{c \in C^{t_1,\ldots,t_n \to cs_k}} \prod_{i=1}^{r_c-1} \mathsf{A}^{t_i}$,

$\bot_{>l}$ is the least element in $\prod_{k=l+1}^{n_{cs}} \prod_{c \in C^{t_1,\ldots,t_n \to cs_k}} \prod_{i=1}^{r_c-1} \mathsf{A}^{t_i}$,

$\bot_{l,<m}$ is the least element in $\prod_{j=1}^{m-1} \prod_{i=1}^{r_{c_j}-1} \mathsf{A}^{t_i}$ for $c_j \in C^{t_1,\ldots,t_n \to cs_l}$

$\bot_{l,>m}$ is the least element in $\prod_{j=m+1}^{m_{cs_l}} \prod_{i=1}^{r_{c_j}-1} \mathsf{A}^{t_i}$ for $c_j \in C^{t_1,\ldots,t_n \to cs_l}$. $\lhd$

Firstly, a new value is created by placing non–recursive arguments into a vector filled with 0. Secondly, the information of arguments which belong to the same equivalence class, contained in the remaining arguments, are taken into account by building the maximum.

Examples:

1. For the list constructors, we have $\mathsf{A}^{\texttt{ListOfInt}} = \mathbb{B}^2$. The escape semantics of the constructors are tuples consisting of an escape tag and the actual abstract function: $\mathfrak{E}[\![\texttt{Nil}]\!] = (\mathfrak{E}[\![\texttt{Nil}]\!]_1, \mathfrak{E}[\![\texttt{Nil}]\!]_2)$ and $\mathfrak{E}[\![\texttt{Cons}]\!] = (\mathfrak{E}[\![\texttt{Cons}]\!]_1, \mathfrak{E}[\![\texttt{Cons}]\!]_2)$. The escape tags are zero and the functional components are:

$$\begin{array}{ll} (\mathfrak{E}[\![\texttt{Nil}]\!])_2 : \mathsf{A}^{\texttt{ListOfInt}} & (\mathfrak{E}[\![\texttt{Cons}]\!])_2 : \mathsf{A}^{\texttt{int}} \times \mathsf{A}^{\texttt{ListOfInt}} \to \mathsf{A}^{\texttt{ListOfInt}} \\ (\mathfrak{E}[\![\texttt{Nil}]\!])_2 = (0,0) & (\mathfrak{E}[\![\texttt{Cons}]\!])_2(a,l) = (0,a) \sqcup l \end{array}$$

2. For the `CTree`/`ITree` constructors, we have $\mathsf{A}^{\texttt{ITree}} = \mathsf{A}^{\texttt{CTree}} = \mathbb{B} \times (\mathbb{B}^4)$. We choose to have the first half of the second component for the `int` layers and the second half for the `char` layers. The abstract functions for the constructors are:

$$\begin{array}{ll} (\mathfrak{E}[\![\texttt{ILeaf}]\!])_2 : \mathsf{A}^{\texttt{int}} \to \mathsf{A}^{\texttt{ITree}} & (\mathfrak{E}[\![\texttt{INode}]\!])_2 : \mathsf{A}^{\texttt{int}} \times \mathsf{A}^{\texttt{ITree}} \times \mathsf{A}^{\texttt{ITree}} \to \mathsf{A}^{\texttt{ITree}} \\ (\mathfrak{E}[\![\texttt{ILeaf}]\!])_2(a) = (0,(a,0,0,0)) & (\mathfrak{E}[\![\texttt{INode}]\!])_2(a,t_1,t_2) = (0,(0,a,0,0)) \sqcup t_1 \sqcup t_2 \\ (\mathfrak{E}[\![\texttt{CLeaf}]\!])_2 : \mathsf{A}^{\texttt{char}} \to \mathsf{A}^{\texttt{CTree}} & (\mathfrak{E}[\![\texttt{CNode}]\!])_2 : \mathsf{A}^{\texttt{char}} \times \mathsf{A}^{\texttt{CTree}} \times \mathsf{A}^{\texttt{CTree}} \to \mathsf{A}^{\texttt{CTree}} \\ (\mathfrak{E}[\![\texttt{CLeaf}]\!])_2(a) = (0,(0,0,a,0)) & (\mathfrak{E}[\![\texttt{CNode}]\!])_2(a,t_1,t_2) = (0,(0,0,0,a)) \sqcup t_1 \sqcup t_2 \end{array}$$

Definition 4.6 (Escape Semantics of Selectors)
Let $c \in C^{t_1,\ldots,t_n \to cs}$ be a constructor such that cs has index l in the order of the sorts and c has index m in the order of constructors. The *escape semantics of* $\texttt{sel}^k\!\!-\!c$ *(*$1 \le k \le n$*)* is defined as

$$\begin{array}{rcl} \mathfrak{E} & : & CSel^{cs \to t} \to \mathsf{A}^{cs \to t} \\ \mathfrak{E}[\![\texttt{sel}^k\!-\!c]\!] & := & (0, \lambda(a). \begin{cases} ((((a)_2)_l)_m)_k & \text{if } k < r_c \\ a & \text{otherwise} \end{cases}) \end{array} \qquad \lhd$$

Remarks:

- To be consistent with the abstract domains, an additional tag is added to the actual function.
- If t_k is a recursive argument, the functional component of a selector is the identity. This corresponds to the pruning of recursion in the abstract domains. Moreover, computation of the abstract value becomes very efficient, especially for *structurally recursive* functions. We discuss this topic in greater detail in Section 4.5.
- For non–recursive arguments, the selector function actually selects the part of its argument which corresponds to the component k. This is done in four stages (see Figure 4.3)
  1. removal of the escape tag of the argument
  2. selection of the value corresponding to the sort cs' in $[cs]$
  3. selection of the value corresponding to the constructor c of result sort cs'
  4. selection of the value corresponding to the component k

$$
\begin{aligned}
a = ((a)_1, (a)_2) &\in \mathbb{B} \times \prod_{cs' \in [cs]} \prod_{c \in C^{t_1,\ldots,t_n \to cs'}} \prod_{i=1}^{r_c - 1} \mathsf{A}^{t_i} \\
(a)_2 = (((a)_2)_1, \ldots, ((a)_2)_l, \ldots, ((a)_2)_{|[cs]|}) &\in \prod_{cs' \in [cs]} \prod_{c \in C^{t_1,\ldots,t_n \to cs'}} \prod_{i=1}^{r_c - 1} \mathsf{A}^{t_i} \\
((a)_2)_l = ((((a)_2)_l)_1, \ldots, (((a)_2)_l)_m, \ldots, (((a)_2)_l)_{|C^{cs'}|}) &\in \prod_{c \in C^{t_1,\ldots,t_n \to cs'}} \prod_{i=1}^{r_c - 1} \mathsf{A}^{t_i} \\
(((a)_2)_l)_m = (((((a)_2)_l)_m)_1, \ldots, ((((a)_2)_l)_m)_k, \ldots, ((((a)_2)_l)_m)_n) &\in \prod_{i=1}^{r_c - 1} \mathsf{A}^{t_i} \\
((((a)_2)_l)_m)_k &\in \mathsf{A}^{t_k}
\end{aligned}
$$

Fig. 4.3: The Four Stages of Non–Recursive Selection

Examples:

1. For the list constructors, we have the abstract functions

$$
\begin{array}{ll}
(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{Cons}]\!])_2 : \mathsf{A}^{\mathtt{ListOfInt}} \to \mathsf{A}^{\mathtt{int}} & (\mathfrak{E}[\![\mathtt{sel}^2\!\!-\mathtt{Cons}]\!])_2 : \mathsf{A}^{\mathtt{ListOfInt}} \to \mathsf{A}^{\mathtt{ListOfInt}} \\
(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{Cons}]\!])_2((x, y)) = y & (\mathfrak{E}[\![\mathtt{sel}^2\!\!-\mathtt{Cons}]\!])_2((x, y)) = (x, y)
\end{array}
$$

   The selection of the head of the list is $(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{Cons}]\!])_2$, which extracts the information for all entries from the abstract value of the list. Note that $(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{Cons}]\!])_2$ is *not* the projection of the first component of its argument. The tail of the list is assumed to have the same escape behaviour as the whole list, and therefore $(\mathfrak{E}[\![\mathtt{sel}^2\!\!-\mathtt{Cons}]\!])_2$ is the identity function.

2. For the non–recursive selectors associated with the `ITree` constructors, we have

$$
\begin{aligned}
&(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{ILeaf}]\!])_2 : \mathsf{A}^{\mathtt{ITree}} \to \mathsf{A}^{\mathtt{int}} \\
&(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{ILeaf}]\!])_2(a_1, (a_2, a_3, a_4, a_5)) = a_2 \\
&(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{INode}]\!])_2 : \mathsf{A}^{\mathtt{ITree}} \to \mathsf{A}^{\mathtt{int}} \\
&(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{INode}]\!])_2(a_1, (a_2, a_3, a_4, a_5)) = a_3
\end{aligned}
$$

   and for the `CTree` constructors

$$
\begin{aligned}
&(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{CLeaf}]\!])_2 : \mathsf{A}^{\mathtt{CTree}} \to \mathsf{A}^{\mathtt{char}} \\
&(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{CLeaf}]\!])_2(a_1, (a_2, a_3, a_4, a_5)) = a_4 \\
&(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{CNode}]\!])_2 : \mathsf{A}^{\mathtt{CTree}} \to \mathsf{A}^{\mathtt{char}} \\
&(\mathfrak{E}[\![\mathtt{sel}^1\!\!-\mathtt{CNode}]\!])_2(a_1, (a_2, a_3, a_4, a_5)) = a_5
\end{aligned}
$$

The next corollary shows that the abstract interpretation of constructors and selectors behave like their concrete counterparts.

COROLLARY 4.3
Let $c \in C^{t_1,\ldots,t_n \to cs}$ be a constructor, $1 \le k \le n$ an index, and $a_i \in \mathsf{A}^{t_i}$ $(1 \le i \le n)$.

1. If $k < r_c$ is the index of a non–recursive argument then

$$(\mathfrak{E}[\![\mathtt{sel}^k\!\!-c]\!])_2((\mathfrak{E}[\![c]\!])_2(a_1, \ldots, a_n)) = a_k$$

2. If $k >= r_c$ is the index of a recursive argument then

$$(\mathfrak{E}[\![\mathtt{sel}^k\!\!-\!c]\!])_2((\mathfrak{E}[\![c]\!])_2(a_1,\ldots,a_n)) = (\mathfrak{E}[\![c]\!])_2(a_1,\ldots,a_n) \qquad \square$$

Note that the escape semantics of both constructors and selector has the property that non–zero entries in the result can only be originated from non–zero entries in arguments.

4.3 Interpretation of (Partial) Applications

An abstract value for a functional type is a tuple of an abstract function and an escape tag. Partial applications of functions again have functional type. Therefore, the result is again a tuple of tag and function. Full application of functions is simply done by dropping the escape tag, and applying the functional component. We define families of auxiliary functions which model this behaviour:

Definition 4.7 (papply, apply)
The family $\mathsf{papply} = \langle \mathsf{papply}_m^{t_1,\ldots,t_n\to t} \mid n \in \mathbb{N}_0, 1 \le m < n,\, t_1,\ldots,t_n,t \in \mathsf{T}(S))\rangle$ is defined as

$$\begin{array}{rcl}\mathsf{papply}_m^{t_1,\ldots,t_n\to t} : \mathsf{A}^{t_1,\ldots,t_n\to t} \times \mathsf{A}^{t_1} \times \cdots \times \mathsf{A}^{t_m} & \to & \mathsf{A}^{t_{m+1},\ldots,t_n\to t} \\ \mathsf{papply}_m^{t_1,\ldots,t_n\to t}((i,f),a_1,\ldots,a_m) & = & (i,\lambda(a_{m+1},\ldots,a_n).f(a_1,\ldots,a_n))\end{array}$$

Here m is the number of arguments given.
The family $\mathsf{apply} = \langle \mathsf{apply}^{t_1,\ldots,t_n\to t} \mid n \in \mathbb{N}_0,\, t_1,\ldots,t_n,t \in \mathsf{T}(S)\rangle$ is defined as

$$\begin{array}{rcl}\mathsf{apply}^{t_1,\ldots,t_n\to t} : \mathsf{A}^{t_1,\ldots,t_n\to t} \times \mathsf{A}^{t_1} \times \cdots \times \mathsf{A}^{t_n} & \to & \mathsf{A}^{t} \\ \mathsf{apply}^{t_1,\ldots,t_n\to t}((i,f),a_1,\ldots,a_n) & = & f(a_1,\ldots,a_n)\end{array} \qquad \triangleleft$$

4.4 The Complete Abstract Interpretation

We are in the position to complete the escape semantics. We start by defining the abstract meaning of basic function.

Definition 4.8 (Escape Semantics of Basic Functions)
Let $f \in BF^{bs_1,\ldots,bs_n\to bs}$ be a basic function. The *escape semantics of* f is defined as

$$\begin{array}{rcl}\mathfrak{E} & : & BF^{bs_1,\ldots,bs_n\to bs} \to \mathsf{A}^{bs_1,\ldots,bs_n\to bs} \\ \mathfrak{E}[\![f]\!] & := & \bot^{bs_1,\ldots,bs_n\to bs}\end{array} \qquad \triangleleft$$

This is reasonable, because basic functions do not create non–zero escape tags in the augmented semantics. Operationally, we can be sure that the result of a basic function is always represented in a newly created heap cell, and, furthermore, basic functions are not created at run-time.

Analogously, we define the abstract meaning of the constructor test functions.

Definition 4.9 (Escape Semantics of Constructor Tests)
Let $c \in C^{t_1,\dots,t_n \to cs}$ be a constructor. The *escape semantics of* $\texttt{is}{-}c$ is defined as

$$\begin{array}{rcl} \mathfrak{E} & : & CTest^{cs \to \texttt{bool}} \to \mathsf{A}^{cs \to \texttt{bool}} \\ \mathfrak{E}[\![\texttt{is}{-}c]\!] & := & \bot^{cs \to \texttt{bool}} \end{array} \qquad \triangleleft$$

In analogy to Ops and $\widehat{\mathsf{Ops}}$, we define the family of escape operations.

Definition 4.10 (Escape Operations)
The family of *escape operations* is defined as $\mathsf{AOps} := \langle \mathsf{AOps}^{t_1,\dots,t_n \to t} \mid t_1,\dots,t_n \to t \in \mathsf{T}(S)\rangle$, where $\mathsf{AOps}^{t_1,\dots,t_n \to t}$ is defined as $\mathsf{AOps}^{t_1,\dots,t_n \to t} := \mathsf{A}^{t_1,\dots,t_n \to t}$. $\triangleleft$

Definition 4.11 (Escape Expression Semantics $\mathfrak{E}$)
Let $e \in E^t$ be an expression, $\chi \in \mathsf{Env}(X,\mathsf{A})$ be an environment for variables, and $\varphi \in \mathsf{Env}(DF,\mathsf{AOps})$ be an environment for defined functions. The *escape semantics of* e *under* χ *and* φ ($\mathfrak{E}[\![e]\!](\chi,\varphi) \in \mathsf{A}^t$) is inductively defined:

- $\mathfrak{E}[\![x]\!](\chi,\varphi) := \chi(x)$ for $x \in X^t$
- $\mathfrak{E}[\![F]\!](\chi,\varphi) := \varphi(F)$ for $F \in DF^t$
- $\mathfrak{E}[\![f]\!](\chi,\varphi) := \mathfrak{E}[\![f]\!]$ for $f \in \Omega^t$
- $\mathfrak{E}[\![(e_0\ e_1\ \dots\ e_m)]\!](\chi,\varphi) :=$
  $$\begin{cases} \mathsf{apply}^{t_1,\dots,t_n \to t_0}(\mathfrak{E}[\![e_0]\!](\chi,\varphi), \mathfrak{E}[\![e_1]\!](\chi,\varphi), \dots, \mathfrak{E}[\![e_m]\!](\chi,\varphi)) & \text{if } m = n \\ \mathsf{papply}_m^{t_1,\dots,t_n \to t_0}(\mathfrak{E}[\![e_0]\!](\chi,\varphi), \mathfrak{E}[\![e_1]\!](\chi,\varphi), \dots, \mathfrak{E}[\![e_m]\!](\chi,\varphi)) & \text{if } 1 \le n < m \end{cases}$$
  for $e_0 \in E^{t_1,\dots,t_n \to t_0}$ and $e_i \in E^{t_i}$ $(1 \le i \le m)$
- $\mathfrak{E}[\![\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](\chi,\varphi) := \mathfrak{E}[\![e_1]\!](\chi,\varphi) \sqcup^t \mathfrak{E}[\![e_2]\!](\chi,\varphi)$ for $e_0 \in E^{\texttt{bool}}$ and $e_1, e_2 \in E^t$ $\triangleleft$

Remarks:

- $f \in \Omega$ handles basic functions (Definition 4.8), constructors (Definition 4.5), selectors (Definition 4.6), and constructor tests (Definition 4.9).
- In a conditional expression '$\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$' the condition e_0 does not contribute to the result, because nothing can escape from there.

Example: For the right hand side

$$\begin{array}{l} e = \texttt{if (is-Nil l) then l'} \\ \quad\ \ \texttt{else (Cons (sel}^1\texttt{-Cons l) (append (sel}^2\texttt{-Cons l) l'))} \end{array}$$

of the `append` definition, a variable environment $\chi = [\texttt{l}/(a_1,a_2), \texttt{l'}/(a_1',a_2')]$, and any environment φ for defined functions, we get

$$
\begin{aligned}
\mathfrak{E}[\![e]\!](\chi,\varphi) &= \mathfrak{E}[\![\texttt{l'}]\!](\chi,\varphi) \sqcup^{\texttt{ListOfInt}} \mathfrak{E}[\![\texttt{(Cons ...)}]\!](\chi,\varphi) \\
&= \chi(\texttt{l'}) \quad \sqcup^{\texttt{ListOfInt}} \quad (0, \mathfrak{E}[\![\texttt{(sel}^1\texttt{-Cons l)}]\!](\chi,\varphi)) \\
&\qquad\qquad \sqcup^{\texttt{ListOfInt}} \quad \mathfrak{E}[\![\texttt{(append (sel}^2\texttt{-Cons l) l')}]\!](\chi,\varphi) \\
&= (a_1', a_2') \quad \sqcup^{\texttt{ListOfInt}} \quad (0, \chi(\texttt{l})_2) \\
&\qquad\qquad \sqcup^{\texttt{ListOfInt}} \quad \varphi(\texttt{append})_2(\mathfrak{E}[\![\texttt{(sel}^2\texttt{-Cons l)}]\!](\chi,\varphi), \mathfrak{E}[\![\texttt{l'}]\!](\chi,\varphi)) \\
&= (a_1', a_2') \sqcup^{\texttt{ListOfInt}} (0, a_2) \sqcup^{\texttt{ListOfInt}} \varphi(\texttt{append})_2(\chi(\texttt{l}), \chi(\texttt{l'})) \\
&= (a_1', a_2' \sqcup^{\texttt{int}} a_2) \sqcup^{\texttt{ListOfInt}} \varphi(\texttt{append})_2((a_1,a_2),(a_1',a_2')) \qquad \diamond
\end{aligned}
$$

Definition 4.12 (Escape Program Semantics $\mathfrak{E}$)
Given a program $P = (F_j(x_{j1},\dots,x_{jn_j}) := e_j \mid 1 \le j \le p)$ with $F_j \in DF^{t_{j1},\dots,t_{jn_j}\to t_j}$, $x_{ji} \in X^{t_{ji}}$, and $e_j \in E^{t_j}$ with variables $\{x_{j1},\dots,x_{jn_j}\}$ for $1 \le j \le p$, the *escape semantics of P* is an environment $\mathfrak{E}[\![P]\!] \in \mathsf{Env}(DF, \mathsf{A})$ defined as

$$\mathfrak{E}[\![P]\!] := [F_1/(0, \mathsf{fix}(\Phi_{\mathfrak{E},P})_1), \dots, F_p/(0, \mathsf{fix}(\Phi_{\mathfrak{E},P})_p)]$$

Here, $\mathsf{fix}(\Phi_{\mathfrak{E},P})$ is the least fixpoint of the transformation $\Phi_{\mathfrak{E},P} : \mathsf{FS}_{\mathfrak{E},P} \to \mathsf{FS}_{\mathfrak{E},P}$ on the function space

$$\mathsf{FS}_{\mathfrak{E},P} := \prod_{j=1}^{p} \left[\mathsf{A}^{t_{j1}} \times \dots \times \mathsf{A}^{t_{jn_j}} \to \mathsf{A}^{t_j}\right]$$

The transformation is defined as

$$
\Phi_{\mathfrak{E},P}(g_1,\dots,g_n) \quad := \quad \begin{pmatrix} \lambda(v_{11},\dots,v_{1n_1}).\mathfrak{E}[\![e_1]\!]([x_{11}/v_{11},\dots,x_{1n_1}/v_{1n_1}],\varphi) \\ \vdots \\ \lambda(v_{p1},\dots,v_{pn_p}).\mathfrak{E}[\![e_p]\!]([x_{p1}/v_{p1},\dots,x_{pn_p}/v_{pn_p}],\varphi) \end{pmatrix}
$$
$$\text{where } \varphi := [F_1/(0,g_1),\dots,F_p/(0,g_p)] \qquad \triangleleft$$

Because the semantic domains are finite cpo's and the expression semantics defines monotonic functions, the fixpoint theorem of Knaster and Tarski (Theorem A.1) guarantees that $\mathsf{fix}(\Phi_{\mathfrak{E},P})$ exists and can be represented in the following way:

$$\mathsf{fix}(\Phi_{\mathfrak{E},P}) = \bigsqcup \{\Phi^i_{\mathfrak{E},P}(\bot^{\mathsf{FS}_{\mathfrak{E},P}}) \mid i \in \mathbb{N}\}$$

Hence, $\mathfrak{E}[\![P]\!]$ is well–defined.

Furthermore, because the semantics domains have only finite ascending chains, we know that the least fixpoint can effectively be computed:

$$\mathsf{fix}(\Phi_{\mathfrak{E},P}) = \Phi^l_{\mathfrak{E},P}(\bot^{t_{1,1},\dots,t_{1,n_1}\to t_1}, \dots, \bot^{t_{p,1},\dots,t_{p,n_p}\to t_p})$$

where l is determined by the maximal chain lengths of all abstract domains involved.

Example: For the `append` program $P_{\texttt{append}}$, we have the following transformation

$$
\begin{aligned}
\Phi_{\mathfrak{E},P_{\texttt{append}}}(g) &= \lambda((a_1,a_2),(a_1',a_2')).\mathfrak{E}[\![\texttt{if ...}]\!]([\texttt{l}/(a_1,a_2), \texttt{l'}/(a_1',a_2')], [\texttt{append}/g]) \\
&= (a_1', a_2' \sqcup^{\texttt{int}} a_2) \sqcup^{\texttt{ListOfInt}} g_2((a_1,a_2),(a_1',a_2'))
\end{aligned}
$$

Successive application of $\Phi_{\mathfrak{E},P_{\texttt{append}}}$ on the least element $\perp = \lambda(x,y).\perp^{\texttt{ListOfInt}}$ of the associated function space $\mathsf{FS}_{\mathfrak{E},P_{\texttt{append}}} = \mathsf{A}^{\texttt{ListOfInt},\texttt{ListOfInt}\to\texttt{ListOfInt}}$ yields the sequence summarised in Table 4.1. The values in the three columns entitled $(\Phi_P^{\mathfrak{E}})_2^i$ contain the values of the functional component of $(\Phi_P^{\mathfrak{E}})^i$ applied to the argument vector $((a_1,a_2),\ (a_1',a_2'))$.

The lines marked with a star are for the *test arguments*, i.e. those which have exactly one non–zero entry. These lines give us information about whether one level of one argument escapes. We can see that the analysis infers that only the first level of the first argument does not escape (line $*_4$).

(a_1,a_2)	(a_1',a_2')	$(\Phi_P^{\mathfrak{E}})_2^0$	$(\Phi_P^{\mathfrak{E}})_2^1$	$(\Phi_P^{\mathfrak{E}})_2^2$	
(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	
(0,0)	(0,1)	(0,0)	(0,1)	(0,1)	$*_1$
(0,0)	(1,0)	(0,0)	(1,0)	(1,0)	$*_2$
(0,0)	(1,1)	(0,0)	(1,1)	(1,1)	
(0,1)	(0,0)	(0,0)	(0,1)	(0,1)	$*_3$
(0,1)	(0,1)	(0,0)	(0,1)	(0,1)	
(0,1)	(1,0)	(0,0)	(1,1)	(1,1)	
(0,1)	(1,1)	(0,0)	(1,1)	(1,1)	
(1,0)	(0,0)	(0,0)	(0,0)	(0,0)	$*_4$
(1,0)	(0,1)	(0,0)	(0,1)	(0,1)	
(1,0)	(1,0)	(0,0)	(1,0)	(1,0)	
(1,0)	(1,1)	(0,0)	(1,1)	(1,1)	
(1,1)	(0,0)	(0,0)	(0,1)	(0,1)	
(1,1)	(0,1)	(0,0)	(0,1)	(0,1)	
(1,1)	(1,0)	(0,0)	(1,1)	(1,1)	
(1,1)	(1,1)	(0,0)	(1,1)	(1,1)	

Tab. 4.1: Fixpoint Computation for `append`

Because we are only interested in the values for the four test arguments, it would be reasonable to compute only those. In the next section, we show that it is indeed possible to do this reduction. ◇

4.5 *Efficient Computation of $\mathfrak{E}$*

Higher-order abstract interpretation is often considered inefficient, since functional types $t_1,\ldots,t_n \to t$ are represented by the set of all monotonic functions from $\mathsf{A}^{t_1}\times\cdots\times\mathsf{A}^{t_n}$ to A^t. Therefore, the straightforward representation of abstract values of this type needs a table of $|\mathsf{A}^{t_1}|\cdot\ldots\cdot|\mathsf{A}^{t_n}|$ elements. Consequently, the worst case complexity for the computation of the fixpoint is quadratic in the number of function definitions, but exponential in the maximal number of arguments.

By using a *test argument* approach, we are able to reduce the representation to a tables of $|\mathsf{A}^{t_1}|+\cdots+|\mathsf{A}^{t_n}|$ elements and to decrease the number of evaluations necessary for the computation of the abstract functions dramatically. Following this approach, the worst case complexity for arbitrary programs is quadratic in both the number of function definitions and maximal number of arguments.

Furthermore, we show that the computation of the fixpoint can be done almost in linear time for realistic programs.

These results are already contained in [Moh95a, Moh95b]. Here, we develop the results using a general theory of complexity of fixpoint computations which is due to Nielson & Nielson [NN92a, NN92b, NN92c].

4.5.1 The Size of a Program

To discuss complexity, we need units of measure for the size of a program. Unlike [Deu97], where the complexity of the escape analysis by Goldberg & Park is studied [GP90, PG92], we consider the "size of the textual representation" to be too coarse to allow precise bounds. In the rest of the section we use the following units as characteristics of a given program $P = (E_1 \dots E_p)$ with equations $E_i = F_i(x_{i,1}, \dots, x_{i,n_i}) = e_i$:

Number of function definitions: p

Number of possible argument combinations: We approximate this by the term s^n, where

- $n := \max\{n_i \mid 1 \leq i \leq p\}$ is the *maximal number of arguments.*
- s is the *size of the largest abstract domain* for all types.

Chain lengths of the abstract domains of the result types: This unit is approximated by the *maximal chain length d* of all abstract domains.

Size of the right hand sides: Since functional programs tend to grow more in "length" than in "width", we consider this as a constant. Moreover, the complexity of a right hand side is mainly determined by the types involved, and hence we assume this factor to be captured by the other units of measure.

In summary, we have the following four numbers:

p: #definitions	s: max. size of abst. dom.
n: max. #args	d: max. chain length in abst. dom.

For our analysis, the abstract domains are isomorphic to bit vectors and hence we have the relation: $s = 2^d$.

4.5.2 The Naïve Fixpoint Computation

The naïve approach is the direct implementation of the Fixpoint Theorem by Knaster and Tarski, i.e. it uses the representation

$$\mathsf{fix}(\Phi) = \bigsqcup_{n \in \mathbb{N}} \Phi^n(\bot)$$

Here all functions are recomputed for all argument combinations until stability. For one function, the number of argument combinations is bound by s^n. Hence, this approach requires the computation of

$$\#\text{eval/iter} = p \cdot s^n$$

values in every iteration. The worst case wrt. to the number of iterations occurs, if in every iteration only a single value changes. The maximum number of changes (until the maximal values in the appropriate domains are reached) is determined by the maximal chain lengths of the result types. Therefore, an upper bound for the number of iterations is

$$\#\text{iter} = p \cdot s^n \cdot d$$

This implies a total of

$$\#\text{eval} = p^2 \cdot s^{2n} \cdot d$$

function evaluations. Hence, the complexity of the naïve approach is quadratic in the number of functions, but exponential in the number of arguments. Note that this is completely independent of the nature of the abstract interpretation.

4.5.3 *Additivity of $\mathfrak{E}$*

The crucial point for the reduction of complexity is that $\mathfrak{E}$ is additive. This property allows the representation of functions without storing their complete graph.

Definition 4.13 (Additive Function)
Let $\langle A;\ \sqcup,\ \sqcap\rangle$, $\langle A';\ \sqcup',\ \sqcap'\rangle$ be complete lattices. A function $f : A \to B$ is *additive* iff for all $a_1, a_2 \in A$ holds

$$f(a_1 \sqcup a_2) = f(a_1) \sqcup' f(a_2) \qquad \lhd$$

Theorem 4.1 ($\mathfrak{E}$ is Additive)
For all programs P and functions $F \in DF^{t_1,\ldots,t_n \to t}$ holds that $((\mathfrak{E}[\![P]\!])(F))_2$ is additive, i.e. for all $a_i, a'_i \in \mathsf{A}^{t_i}$ $(1 \leq i \leq n)$ holds:

$$\begin{aligned} & ((\mathfrak{E}[\![P]\!])(F))_2((a_1,\ldots,a_n) \sqcup^{t_1\times\cdots\times t_n} (a'_1,\ldots,a'_n)) \\ = \ & ((\mathfrak{E}[\![P]\!])(F))_2(a_1,\ldots,a_n) \sqcup^t ((\mathfrak{E}[\![P]\!])(F))_2(a'_1,\ldots,a'_n) \end{aligned}$$

where $\sqcup^{t_1\times\cdots\times t_n} \subseteq (\mathsf{A}^{t_1} \times \cdots \times \mathsf{A}^{t_n})^2$ is defined componentwise. □

Proof It is obvious from Definitions 4.5, 4.6, 4.8 and 4.9 that intrinsic functions are additive. By induction, we can prove the same for expressions of functional type, if all entries in the function environment are also additive. Since additivity is a continuous property, we then have the result for the fixpoint. q.e.d.

However, we do not have *completely additive functions*, i.e. $\mathfrak{E}[\![P]\!](F)$ does not need to be strict. Consider the program P with the function definition `F x y = x`. If we consider the partial application `(F z)`, we have $\mathfrak{E}[\![\texttt{(F z)}]\!](\chi, \mathfrak{E}[\![P]\!]) = \lambda x.\chi(\texttt{z})$. If $\chi(\texttt{z}) \neq \bot$ then this result is not a strict function.

The next step is to find an appropriate set of abstract elements, such that every element is representable by these.

DEFINITION 4.14 (JOIN–IRREDUCIBLE ELEMENTS)
Let $\langle A;\ \sqcup,\ \sqcap\rangle$ be a complete lattice. An element $a \in A$ is *join–irreducible* iff for all $a_1, a_2 \in A$ $a = a_1 \sqcup a_2$ implies that $a = a_1$ or $a = a_2$. $\lhd$

It is easy to see that every element a of a complete lattice can be represented by the least upper bound of a set of join–irreducible elements L, i.e. we have $a = \bigsqcup L$. Hence, given an additive function $f : A \to B$, we have $f(a) = f(\bigsqcup L) = \bigsqcup f(L)$ and therefore it is sufficient to know the values of f for the join–irreducible elements of A.

Note that the least element $\perp$ of a lattice is always join–irreducible. In contrast to the work by Nielson & Nielson, we cannot ignore this element, since we do not have completely additive functions. The other join–irreducible elements of A^t are essentially those which contain only one non–zero bit.

Examples:

1. For a function $f : \mathsf{A}^{\texttt{ListOfInt}} \to \mathsf{A}^t$ we need $|\mathsf{A}^{\texttt{ListOfInt}}| = 2^2 = 4$ elements of A^t to represent the graph. The three join–irreducible elements of $\mathsf{A}^{\texttt{ListOfInt}}$ are

$$\{(1,0),(0,1),(0,0)\}$$

   Hence, we can represent the graph by using only the three values of f at those positions. In the corresponding Hasse–diagrams, we can see that the test values are the lowest strata plus the bottom element in the diagram (Figure 4.4(a)).

2. We consider the type of nested lists of integers:

```
ListOfList ::= LNil | LCons ListOfInt ListOfList
```

   For a function $f : \mathsf{A}^{\texttt{ListOfList}} \to \mathsf{A}^t$ we need four elements instead of eight (see Figure 4.4(b)).

3. A higher–order function $f : \mathsf{A}^{\texttt{int}\to\texttt{int}} \to \mathsf{A}^t$ has the following test arguments:

$$\left\{\left(0, \begin{matrix}0\mapsto 1\\ 1\mapsto 0\end{matrix}\right), \left(0, \begin{matrix}0\mapsto 0\\ 1\mapsto 1\end{matrix}\right), \left(1, \begin{matrix}0\mapsto 0\\ 1\mapsto 0\end{matrix}\right), \left(0, \begin{matrix}0\mapsto 0\\ 1\mapsto 0\end{matrix}\right)\right\}$$

   In a strict sense, we do not need the elements set in *italic* in Figure 4.4(c) since these functions are not additive. We could use the test argument $(0, {}^{0\mapsto 1}_{1\mapsto 1})$ instead of $(0, {}^{0\mapsto 1}_{1\mapsto 0})$.

By using the set of join–irreducible elements as test arguments, we observe a significant reduction of complexity: Instead of computing the values of functions for all inputs, we compute only the values for test arguments. The next result uses that A^t is also a distributive lattice, which allows us to approximate the number of join–irreducible elements by the maximal chain length.

LEMMA 4.3
Let d be the maximal chain length in A^t. For the number J of join–irreducible elements of A^t holds: $J \leq d + 1$. $\square$

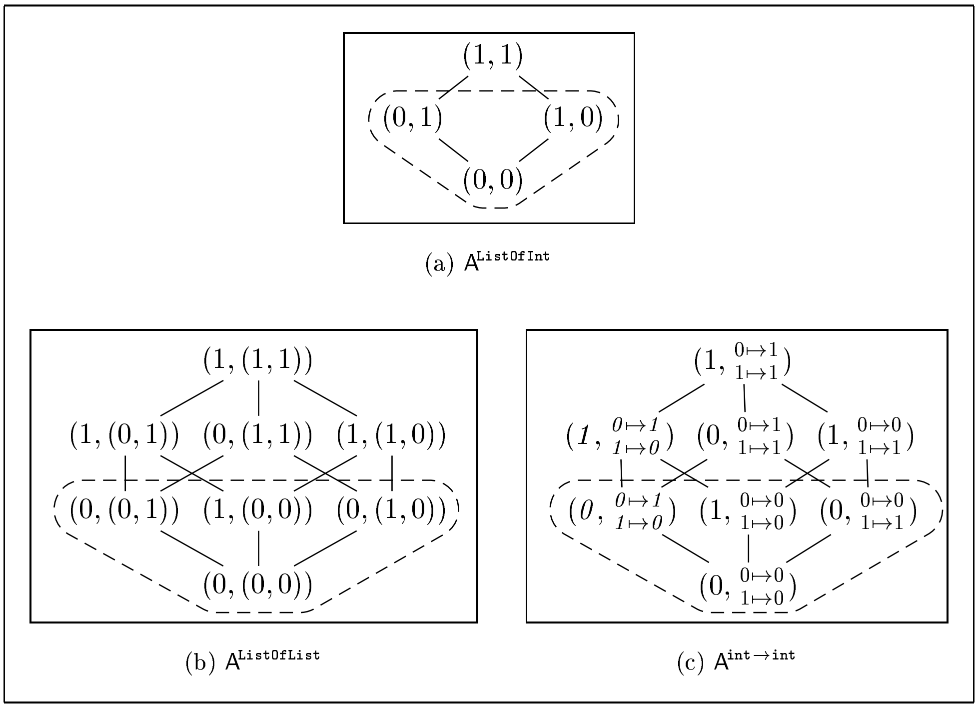


Fig. 4.4: Hasse Diagrams of Abstract Domains

PROOF [Grä71, p. 73] q.e.d.

This allows us to make new approximations. The number of arguments to consider for each function is now bound by $n \cdot (d+1)$. Since we only need to evaluate the functions for the test arguments in each iteration, the number of evaluations is linear in the number of arguments:

$$\#\text{eval/iter} = p \cdot n \cdot (d+1)$$

Also the number of iterations reduces, because less changes can occur:

$$\#\text{iter} = p \cdot n \cdot (d+1)^2$$

This implies a total of

$$\#\text{eval} = p^2 \cdot n^2 \cdot (d+1)^3$$

function evaluations. Now the complexity is *quadratic both in the number of functions and in the number of arguments.*

4.5.4 *Structurally Recursive Functions*

Realistic programs, however, seldom reach this worst case, since the function definitions are not that enigmatic. Especially *structurally recursive* functions, like the `append` function, are interesting. Looking at the corresponding transformation, we can observe that the value of

$\mathfrak{E}[\![\texttt{append}]\!]$ for an argument (a_1, a_2) in the $i+1$-th iteration depends only on the value for (a_1, a_2) in the i-th iteration and not on the value for any other argument.

$$\begin{aligned}
&\mathfrak{E}[\![\texttt{append}]\!](a_1, a_2) \\
&\quad = \mathfrak{E}[\![\texttt{Nil}]\!] \sqcup \mathfrak{E}[\![\texttt{Cons}]\!](\texttt{sel}^1\!\!-\!\texttt{Cons}(a_1), \mathfrak{E}[\![\texttt{append}]\!](\texttt{sel}^2\!\!-\!\texttt{Cons}(a_1), a_2)) \\
&\quad = \mathfrak{E}[\![\texttt{Cons}]\!](\texttt{sel}^1\!\!-\!\texttt{Cons}(a_1), \mathfrak{E}[\![\texttt{append}]\!](a_1, a_2)) \\
&\quad = (0, \texttt{sel}^1\!\!-\!\texttt{Cons}(a_1)) \sqcup \mathfrak{E}[\![\texttt{append}]\!](a_1, a_2)
\end{aligned}$$

The reason is that the abstract interpretation of $\mathfrak{E}[\![\texttt{sel}^2\!\!-\!\texttt{Cons}]\!]$ is the identity function, corresponding to the axiom that all constructors of the same level behave in the same way. Functions which are defined via structural recursion always have this property.

The effect on the complexity of the computation is enormous. Because the value for each argument is independent from the other values, the computation of the fixpoint is stable after at most d steps. For a program where all definitions have this form, the overall number of function evaluations drops to

$$\#\text{eval} = p \cdot n \cdot (d+1)^3$$

which is linear both wrt. the number of definitions and arguments.

We can also exploit this property for programs P where not all definitions have this form. There we can decompose P into P_{sr} and P_{nsr} such that P_{sr} contains only structurally recursive definitions. By using *chaotic fixpoint computation* [CC77, VWL94] we can find the fixpoint for the structural recursive part P_{sr} first, and then compute the rest.

4.6 Summary

We have presented the escape semantics $\mathfrak{E}$ of the language F, an abstract interpretation which allows us to approximate the escape behaviour of programs. The design criteria for the abstraction were based on (1) the removal of concrete values from data structures, and (2) the compression of unbounded or infinite values to finite abstract values. The basic concept of the compression was the assumption that all constructors of the same level behave in the same way. Furthermore, we have demonstrated that the design of the abstract interpretation results in quadratic complexity for the computation of the information.

5. Denotational Correctness

In this chapter we show that the abstract interpretation $\mathfrak{E}$ provides safe approximations of the escape behaviour of programs. Safeness in this context means that if $\mathfrak{E}$ judges that no escape occurs then we can be sure that this is true. On the other hand, if the result of $\mathfrak{E}$ indicates that something escapes then there may be no escape. Hence, $\mathfrak{E}$ gives us a correct, but not complete approximation of escape behaviour. But since escaping is a non–trivial semantic property, it is in general not possible to define a (decidable) abstract interpretation, which is both correct and complete.

Comparing this situation with that in strictness analysis [Myc80, BHA86a], we see that the terminology is not consistent. While strictness analysis is a safe approximation of strictness, escape analysis is a save approximation of non–escaping.

5.1 Relating Abstract and Concrete Domains

We need some means to relate the abstract domains A^t and the concrete domains $\widehat{\mathsf{CT}}^t$. The first rigorous framework for this task was developed in [CC77]. There, the correspondence between a concrete domain D_C and an abstract domain D_A is established through a *Galois connection* $(\mathsf{Abs}, \mathsf{Conc})$, i.e. an adjoined pair of abstraction and concretisation functions with the properties:

$$\begin{array}{ll} \mathsf{Abs} : D_C \to D_A \qquad & \mathsf{Conc} : D_A \to D_C \\ \mathsf{Abs} \circ \mathsf{Conc} = \mathsf{id} \qquad & \mathsf{Conc} \circ \mathsf{Abs} \geq \mathsf{id} \end{array}$$

Intuitively, the properties ensure that concretisation does not loose information, whereas abstraction may loose information by increasing the concrete value. An abstract interpretation is safe wrt. a concrete interpretation in this framework iff the following diagram holds:

$$\begin{tikzcd}
a_1 \arrow[r, "\text{abstract computation}"] & a_2 \arrow[d, "\mathsf{Conc}"] \\
& \rotatebox{90}{$\leq$} \\
v_1 \arrow[uu, "\mathsf{Abs}"] \arrow[r, "\text{concrete computation}"'] & v_2
\end{tikzcd}$$

Unfortunately, we cannot use this approach directly, because it is not possible to define a function $\mathsf{Conc} : \mathsf{A} \to \widehat{\mathsf{CT}}$ with the above properties. Clearly, *many* augmented values relate to a *single* abstract value, e.g. $(1, 0) \in \mathsf{A}^{\mathtt{ListOfInt}}$ represents all list where at least one constructor is tagged with '1'. This problem did not occur in [CC77] because they used a *collecting semantics*, i.e. one concrete value represented *all computations*.

However, we can use the modified approach described in [AH87b, BHA86a, BHA86b] where we only need an abstraction function. An abstract interpretation is safe wrt. a concrete interpretation in this framework iff the following diagram holds:

$$
\begin{tikzcd}
a_1 \arrow[r, "\text{abstract computation}"] & a_2 \\
 & \rotatebox{90}{$\leq$} \\
v_1 \arrow[uu, "\alpha"] \arrow[r, "\text{concrete computation}"'] & v_2 \arrow[u, "\alpha"']
\end{tikzcd}
$$

This approach is equivalent to the approach of [CC77] if we move to a collecting semantics and define

$$
\begin{array}{ll}
\mathsf{Abs} : \mathfrak{P}(\widehat{\mathsf{CT}}) \to \mathfrak{P}(\mathsf{A}) & \mathsf{Conc} : \mathfrak{P}(\mathsf{A}) \to \mathfrak{P}(\widehat{\mathsf{CT}}) \\
\mathsf{Abs}(\widehat{V}) := \alpha(\widehat{V}) & \mathsf{Conc}(A) := \alpha^{-1}(A)
\end{array}
$$

Definition 5.1 (Abstraction Function α)

1. The abstraction function $\alpha : \widehat{\mathsf{CT}} \to \mathsf{A}$ is defined as:

   (a) $\alpha((b, v)) = b$ for all $(b, v) \in \widehat{\mathsf{CT}}^{bs}$, $bs \in BS$

   (b) $\alpha(\widehat{v}) = \bigsqcup^{cs}\{\alpha_p(p) \mid p \in \widehat{v}\}$ for all $\widehat{v} \in \widehat{\mathsf{CT}}^{cs}$, $cs \in CS$
   where $\alpha_p : \widehat{\mathsf{PT}}^{cs} \to \mathsf{A}^{cs}$ is defined as

   i. $\alpha_p((b, \perp^{cs})) := (b, (0, \ldots, 0)) \in \mathsf{A}^{cs}$

   ii. $c \in C^{t_1,\ldots,t_n \to cs}$, $b \in \mathbb{B}$, and for $1 \leq i \leq n$: $\begin{cases} w_i \in \widehat{\mathsf{PT}}^{t_i},\ \alpha_i = \alpha_p & \text{if } t_i \in CS \\ w_i \in \widehat{\mathsf{CT}}^{t_i},\ \alpha_i = \alpha & \text{if } t_i \notin CS \end{cases}$
   $\alpha_p((b, c(w_1, \ldots, w_n))) := (b, (0, \ldots, 0)) \sqcup^{cs} \mathfrak{E}[\![c]\!](\alpha_1(w_1), \ldots, \alpha_n(w_n))$

   (c) $\alpha((b, f)) := (b, \alpha(f))$ where

   $$\alpha(f) = \lambda(a_1, \ldots, a_n). \bigsqcup^{t}\{\alpha(f(\widehat{v}_1, \ldots, \widehat{v}_n)) \mid \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i},\ \alpha(\widehat{v}_i) \preceq^{t_i} a_i,\ 1 \leq i \leq n\}$$

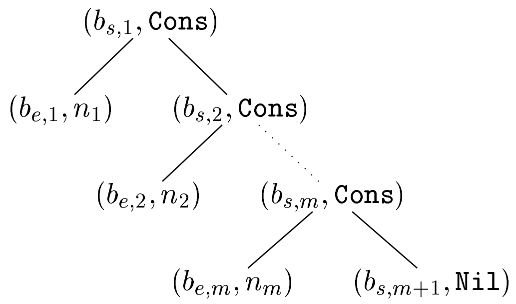
   for all $(b, f) \in \widehat{\mathsf{CT}}^{t_1,\ldots,t_n \to t}$, $t_1, \ldots, t_n \to t \in \mathsf{T}(S)$

2. The abstraction function for environments $\alpha : \mathsf{Env}(X, \widehat{\mathsf{CT}}) \to \mathsf{Env}(X, \mathsf{A})$ is defined componentwise: $(\alpha(\widehat{\beta}))(x) := \alpha(\widehat{\beta}(a))$ for all $x \in X$. $\lhd$

Remarks:

- α essentially removes the concrete values and keeps the escape tags.
- For data structures, we use $\mathfrak{E}[\![c]\!]$ to remove recursion.
- For functions, we compute the worst case behaviour since escaping in the augmented domain may depend on the actual parameters. Monotonicity implies that the condition $\alpha(\widehat{v}_i) \preceq^{t_i} a_i$ can be replaced by $\alpha(\widehat{v}_i) = a_i$ if we prove that α is surjective.

Example: Consider the augmented values for lists over integers $[n_1, \ldots, n_m]$ which can be pictured in the following way



Abstracting such a value yields

$$\begin{aligned} & \alpha((b_{s,1}, \texttt{Cons}((b_{e,1}, n_1), \ldots \texttt{Cons}((b_{e,m}, n_m), (b_{s,m}, \texttt{Nil}))))) \\ = \quad & (b_{s,1} \vee \cdots \vee b_{s,m}, b_{e,1} \vee \cdots \vee b_{e,m}) \end{aligned} \qquad \diamond$$

Lemma 5.1 (Properties of α)
α is surjective: $\alpha(\widehat{\mathsf{CT}}) = \mathsf{A}$.

α is monotonic: $\alpha(\widehat{v}_1) \preceq^t \alpha(\widehat{v}_2)$ for all $\widehat{v}_1, \widehat{v}_2 \in \widehat{\mathsf{CT}}^t$ with $\widehat{v}_1 \,\widehat{\leq}^t\, \widehat{v}_2$.

α is distributive: $\alpha(f(\widehat{v}_1, \ldots, \widehat{v}_n)) \preceq^t (\alpha(f))(\alpha(\widehat{v}_1), \ldots, \alpha(\widehat{v}_n))$ for all $(b, f) \in \widehat{\mathsf{CT}}^{t_1,\ldots,t_n \to t}$, $\widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}$, and $1 \leq i \leq n$. $\square$

5.2 Denotational Safeness

The abstraction function α is the key for the notion of safeness used in this chapter.

Definition 5.2 (Safe Approximation)
Let $a \in \mathsf{A}^t$ be an abstract value and $\widehat{v} \in \widehat{\mathsf{CT}}^t$ be an augmented value. If $\alpha(\widehat{v}) \preceq^t a$ then we say that a is a *safe approximation* of $\widehat{v}$. $\lhd$

Our next aim is to show that a program's escape semantics is a safe approximation of its augmented semantics. Corresponding to the structure of programs and expressions, we start with intrinsic function.

Lemma 5.2
For all intrinsics $f \in \Omega^{t_1,\ldots,t_n \to t}$ holds that $\mathfrak{E}[\![f]\!]$ is a safe approximation of $\widehat{\mathfrak{M}}[\![f]\!]$. $\square$

Proof We distinguish four cases:

1. $f = bf \in BF$

$$\begin{aligned} \alpha(\widehat{\mathfrak{M}}[\![bf]\!]) &= \alpha(\oplus(\mathfrak{M}[\![bf]\!])) \\ &= (0, \lambda(\overline{a}).\bigsqcup\nolimits^{t_0}\{\alpha(\oplus(\mathfrak{M}[\![bf]\!](\overline{\ominus(\widehat{v})}))) \mid \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i},\ \alpha(\widehat{v}_i) \preceq^{t_i} a_i,\ 1 \leq i \leq n\}) \end{aligned}$$

$$
\begin{aligned}
&= (0, \lambda(\overline{a}).\bigsqcup^{t_0}\{\alpha((0, \mathfrak{M}[\![bf]\!](\overline{\ominus(\widehat{v})}))) \mid \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i},\ \alpha(\widehat{v}_i) \preceq^{t_i} a_i,\ 1 \leq i \leq n\}) \\
&= (0, \lambda(\overline{a}).\bigsqcup^{t_0}\{0 \mid \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i},\ \alpha(\widehat{v}_i) \preceq^{t_i} a_i,\ 1 \leq i \leq n\}) \\
&= (0, \lambda(\overline{a}).0) \\
&= \perp^{t_1,\ldots,t_n \to t} \\
&= \mathfrak{E}[\![bf]\!]
\end{aligned}
$$

2. $f = c \in C$

$$
\begin{aligned}
\alpha(\widehat{\mathfrak{M}}[\![c]\!]) &= (0, \lambda(a_1, \ldots, a_n)(0, \perp_1, \ldots, \perp_{l-1}, \\
&\qquad (\perp_{l,1}, \ldots, \perp_{l,m-1}, (a_1, \ldots, a_{r_c-1}), \perp_{l,m+1}, \ldots, \perp_{l,m_{cs_l}}) \\
&\qquad \perp_{l+1}, \ldots, \perp_{n_{cs}}) \sqcup \textstyle\bigsqcup_{i=r_c}^{n} a_i) \\
&= \mathfrak{E}[\![c]\!]
\end{aligned}
$$

3. $f = \mathtt{sel}\_^{j}\_c \in CSel$

Let $c \in C^{t_1,\ldots,t_n \to cs}$ such that cs has index l in the order of the sorts and c has index m in the order of constructors.

$$
\begin{aligned}
&\alpha(\widehat{\mathfrak{M}}[\![\mathtt{sel}\_^{j}\_c]\!]) \\
&= \alpha((0, \lambda(\widehat{v}).\begin{cases} \widehat{v}_j & \text{if } \widehat{v} = (b, c(\widehat{v}_1, \ldots, \widehat{v}_m)) \text{ and } 1 \leq j \leq m \\ \widehat{\perp}^{t_j} & \text{otherwise} \end{cases})) \\
&= (0, \lambda(a).\bigsqcup^{t_j}\{\alpha(\begin{cases} \widehat{v}_j & \widehat{v} = (b, c(\widehat{v}_1, \ldots, \widehat{v}_m)),\ 1 \leq j \leq m \\ \widehat{\perp}^{t_j} & \text{otherwise} \end{cases}) \mid \widehat{v} \in \widehat{\mathsf{CT}}^{cs},\ \alpha(\widehat{v}) \preceq^{cs} a\}) \\
&\overset{(*)}{\preceq^{t}} (0, \lambda(a).\begin{cases} ((((a)_2)_l)_m)_k & \text{if } k < r_c \\ a & \text{otherwise} \end{cases}) \\
&= \mathfrak{E}[\![\mathtt{sel}\_^{j}\_c]\!]
\end{aligned}
$$

To verify $(*)$ we can assume that $1 \leq j \leq m$, because $\alpha(\widehat{\perp}^{t_j}) = \perp^{t_j}$. What remains to be shown is that

$$
\begin{aligned}
&\bigsqcup^{t_j}\{\alpha(\widehat{v}_j) \mid \widehat{v} \in \widehat{\mathsf{CT}}^{cs},\ \alpha(\widehat{v}) \preceq^{cs} a,\ \widehat{v} = (b, c(\widehat{v}_1, \ldots, \widehat{v}_m))\} \\
&\qquad\qquad \preceq^{t_j} \quad \begin{cases} ((((a)_2)_l)_m)_k & \text{if } k < r_c \\ a & \text{otherwise} \end{cases}
\end{aligned}
$$

It suffices to prove this is true for each element of the chain, i.e. let $\widehat{v} \in \widehat{\mathsf{CT}}^{cs}$ with $\alpha(\widehat{v}) \preceq^{cs} a$ and $\widehat{v} = (b, c(\widehat{v}_1, \ldots, \widehat{v}_m))$.

$$
\begin{aligned}
\alpha(\widehat{v}) &= \bigsqcup^{cs}\{\alpha_p(p) \mid p \in \widehat{v}\} \\
&= \bigsqcup^{cs}\{(b, (0, \ldots, 0)) \sqcup^{cs} \mathfrak{E}[\![c]\!](\alpha_1(w_1), \ldots, \alpha_n(w_n)) \mid (b, c(w_1, \ldots, w_n)) \in \widehat{v}\} \\
&= (b, (0, \ldots, 0)) \sqcup^{cs} \bigsqcup^{cs}\{\mathfrak{E}[\![c]\!](\alpha_1(w_1), \ldots, \alpha_n(w_n)) \mid (b, c(w_1, \ldots, w_n)) \in \widehat{v}\}
\end{aligned}
$$

Let $(b, c(w_1, \ldots, w_n)) \in \widehat{v}$:

$$
\begin{aligned}
&\mathfrak{E}[\![c]\!](\alpha_1(w_1), \ldots, \alpha_n(w_n)) \\
=\ &(0, \perp_{<l}, (\perp_{l,<m}, (\alpha_1(w_1), \ldots, \alpha_{r_c-1}(w_{r_c-1})), \perp_{l,>m}), \perp_{>l}) \sqcup \bigsqcup_{i=r_c}^{n} \alpha_i(w_i)
\end{aligned}
$$

Hence, we have with the abbreviation $\bigsqcup^{cs}_{\in\widehat{v}} . = \bigsqcup^{cs}\{. \mid (b, c(w_1, \ldots, w_n)) \in \widehat{v}\}$

$$\begin{array}{rcl}\alpha(\widehat{v}) & = & (b, \bot_{<l}, (\bot_{l,<m}, (\bigsqcup^{cs}_{\in\widehat{v}} \alpha_1(w_1), \ldots, \bigsqcup^{cs}_{\in\widehat{v}} \alpha_{r_c-1}(w_{r_c-1})), \bot_{l,>m}), \bot_{>l}) \\ & & \sqcup \bigsqcup^{cs}_{\in\widehat{v}} \bigsqcup_{i=r_c}^{n} \alpha_i(w_i) \\ & \preceq^{t_j} & a\end{array}$$

We distinguish two cases:

(a) If $k < r_c$ then we have $\alpha(\widehat{v}_j) \preceq^{t_j} \bigsqcup^{cs}_{\in\widehat{v}} \alpha_i(w_i) \preceq^{t_j} ((((a)_2)_l)_m)_k$.

(b) Otherwise, we have $\alpha(\widehat{v}_j) \preceq^{t_j} \bigsqcup^{cs}_{\in\widehat{v}} \bigsqcup^{n}_{i=r_c} \alpha_i(w_i) \preceq^{t_j} a$.

4. $f = \texttt{is}{-}c \in CTest$

$$\begin{array}{l}\alpha(\widehat{\mathfrak{M}}[\![\texttt{is}{-}c]\!]) \\ = \alpha((0, \lambda(\widehat{v}). \begin{cases} (0, \texttt{true}) & \text{if } \widehat{v} = (b, c(\widehat{v}_1, \ldots, \widehat{v}_m)) \\ \widehat{\bot}^{\texttt{bool}} & \text{if } \widehat{v} = \widehat{\bot}^{cs} \\ (0, \texttt{false}) & \text{otherwise} \end{cases})) \\ = (b, \lambda(a). \bigsqcup^{\texttt{bool}}\{\alpha((0, \begin{cases} (0, \texttt{true}) & \text{if } \widehat{v} = (b, c(\widehat{v}_1, \ldots, \widehat{v}_m)) \\ \widehat{\bot}^{\texttt{bool}} & \text{if } \widehat{v} = \widehat{\bot}^{cs} \\ (0, \texttt{false}) & \text{otherwise} \end{cases})) \mid \widehat{v} \in \widehat{\mathsf{CT}}^{cs},\ \alpha(\widehat{v}) \preceq^{cs} \\ \quad a\}) \\ = (0, \lambda(a). \bigsqcup^{\texttt{bool}}\{0 \mid \widehat{v} \in \widehat{\mathsf{CT}}^{cs},\ \alpha(\widehat{v}) \preceq^{cs} a\}) \\ = (0, \lambda(a).0) \\ = \bot^{cs \to \texttt{bool}} \\ = \mathfrak{E}[\![\texttt{is}{-}c]\!]\end{array}$$

q.e.d.

THEOREM 5.1
Let $e \in E^t$, $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$, and $\widehat{\sigma} \in \mathsf{Env}(DF, \widehat{\mathsf{Ops}})$: $\mathfrak{E}[\![e]\!](\alpha(\widehat{\beta}), \alpha(\widehat{\sigma}))$ is a safe approximation of $\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma})$). □

PROOF Induction on e

$e = x \in X$: $\alpha(\widehat{\mathfrak{M}}[\![x]\!](\widehat{\beta}, \widehat{\sigma})) = \alpha(\widehat{\beta}(x)) = (\alpha(\widehat{\beta}))(x) = \mathfrak{E}[\![x]\!](\alpha(\widehat{\beta}), \alpha(\widehat{\sigma}))$

$e = F \in DF$: $\alpha(\widehat{\mathfrak{M}}[\![F]\!](\widehat{\beta}, \widehat{\sigma})) = \alpha(\widehat{\sigma}(F)) = (\alpha(\widehat{\sigma}))(F) = \mathfrak{E}[\![F]\!](\alpha(\widehat{\beta}), \alpha(\widehat{\sigma}))$

$e = f \in \Omega$: Lemma 5.2

$e = (e_0\ e_1\ \ldots\ e_m)$: By induction, we have

$$\alpha(\widehat{\mathfrak{M}}[\![e_i]\!](\widehat{\beta}, \widehat{\sigma})) \preceq^{t_i} \mathfrak{E}[\![e_i]\!](\alpha(\widehat{\beta}), \alpha(\widehat{\sigma})) \quad \text{for } 0 \leq i \leq m$$

Let $t_1, \ldots, t_n \to t_0$ be the type of e_0, $(b, f) = \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta}, \widehat{\sigma})$, $(i, f_a) = \mathfrak{E}[\![e_0]\!](\alpha(\widehat{\beta}), \alpha(\widehat{\sigma}))$. Two cases may occur: either $m = n$, i.e. the application is saturated, or $m < n$, i.e. we have a partial application of e_0.

$$
\begin{aligned}
n = m{:}\quad & \alpha(\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta},\widehat{\sigma})) \\
&= \alpha(f(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta},\widehat{\sigma}),\ldots,\widehat{\mathfrak{M}}[\![e_n]\!](\widehat{\beta},\widehat{\sigma}))) \\
&\preceq^t (\alpha(f))(\alpha(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta},\widehat{\sigma})),\ldots,\alpha(\widehat{\mathfrak{M}}[\![e_n]\!](\widehat{\beta},\widehat{\sigma}))) \\
&\preceq^t f_a(\mathfrak{E}[\![e_1]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma})),\ldots,\mathfrak{E}[\![e_n]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma}))) \\
&= \mathsf{apply}^{t_1,\ldots,t_n\to t_0}(\mathfrak{E}[\![e_0]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma})),\mathfrak{E}[\![e_1]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma})),\ldots,\mathfrak{E}[\![e_n]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma}))) \\
&= \mathfrak{E}[\![e]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma}))
\end{aligned}
$$

$m < n$: analogously

$e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: By induction, we have

$$\alpha(\widehat{\mathfrak{M}}[\![e_i]\!](\widehat{\beta},\widehat{\sigma})) \preceq^{t_i} \mathfrak{E}[\![e_i]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma})) \quad \text{for } 1 \le i \le 2$$

$$
\begin{aligned}
\alpha(\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta},\widehat{\sigma})) &= \begin{cases} \alpha(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta},\widehat{\sigma})) & \text{if } \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta},\widehat{\sigma}) = (b,\texttt{true}), b \in \mathbb{B} \\ \alpha(\widehat{\bot}^t) & \text{if } \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta},\widehat{\sigma}) = \widehat{\bot}^{\texttt{bool}} \\ \alpha(\widehat{\mathfrak{M}}[\![e_2]\!](\widehat{\beta},\widehat{\sigma})) & \text{if } \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta},\widehat{\sigma}) = (b,\texttt{false}), b \in \mathbb{B} \end{cases} \\
&\preceq^t \alpha(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta},\widehat{\sigma})) \sqcup^t \alpha(\widehat{\bot}^t) \sqcup^t \alpha(\widehat{\mathfrak{M}}[\![e_2]\!](\widehat{\beta},\widehat{\sigma})) \\
&\preceq^t \mathfrak{E}[\![e_1]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma})) \sqcup^t \bot^t \sqcup^t \mathfrak{E}[\![e_2]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma})) \\
&= \mathfrak{E}[\![e]\!](\alpha(\widehat{\beta}),\alpha(\widehat{\sigma}))
\end{aligned}
$$

q.e.d.

Corollary 5.1 (Safeness of $\mathfrak{E}$)
For all programs P and functions $F \in DF^t$ holds that $(\mathfrak{E}[\![P]\!])(F)$ is a safe approximation of $(\widehat{\mathfrak{M}}[\![P]\!]))(F)$. □

5.3 Void Abstract Values

Safety is not the only important property of abstract interpretations. An abstract interpretation which assigns the top element of the abstract domains to all expression is safe by definition. However, it is not *meaningful* since it never allows to determine that no escaping occurs. The escape analysis $\mathfrak{E}$ is meaningful, as we have seen in the last chapter.

Example: For the `append` program $P_{\texttt{append}}$, we have

$$\mathfrak{E}[\![\texttt{(append l l')}]\!]([\texttt{l}/(1,0),\texttt{l'}/(0,0)]) = (0,0)$$

which means that for all $\widehat{v},\widehat{v}' \in \widehat{\mathsf{CT}}^{\texttt{ListOfInt}}$ with $\alpha(\widehat{v}) = (1,0)$ and $\alpha(\widehat{v'}) = (0,0)$ holds

$$\alpha(\widehat{\mathfrak{M}}[\![\texttt{(append l l')}]\!]([\texttt{l}/\widehat{v},\texttt{l'}/\widehat{v'}])) \preceq^{\texttt{ListOfInt}} (0,0)$$

and hence

$$\alpha(\widehat{\mathfrak{M}}[\![\texttt{(append l l')}]\!]([\texttt{l}/\widehat{v},\texttt{l'}/\widehat{v'}])) = \bot^{\texttt{ListOfInt}}$$

Furthermore, it is obvious that $\alpha(\widehat{v}) = \bot^{\texttt{ListOfInt}}$ iff $\widehat{v}$ has void augmentation. Consequently, we know that the spine of the first list does not escape from the expression. ◇

What remains to be done is to characterise those abstract values a with $\alpha(\widehat{v})$ such that $\widehat{v}$ has void augmentation. For basic types and constructed types it is obvious that augmented values with void augmentation have $\perp$ as abstraction. Functional values, however, may have void augmentation although their abstraction is not equal to $\perp$.

Example: Consider the value $(0, \mathsf{id}) \in \widehat{\mathsf{CT}}^{\mathtt{int}\to\mathtt{int}}$ which has void augmentation. However,

$$\begin{aligned}
\alpha((0,\mathsf{id})) &= (0, \lambda(a). \bigsqcup^{\mathtt{int}} \{\alpha(\mathsf{id}(\widehat{v})) \mid \widehat{v} \in \widehat{\mathsf{CT}}^{\mathtt{int}},\ \alpha(\widehat{v}) \preceq^{\mathtt{int}} a\}) \\
&= (0, \lambda(a). \bigsqcup^{\mathtt{int}} \{\alpha(\widehat{v}) \mid \widehat{v} \in \widehat{\mathsf{CT}}^{\mathtt{int}},\ \alpha(\widehat{v}) \preceq^{\mathtt{int}} a\}) \\
&= (0, \lambda(a).a) \\
&\neq (0, \lambda(a).0) \\
&= \perp^{\mathtt{ListOfInt}}
\end{aligned}$$ ◇

The reason is simply that in addition to the actual escape tags, the abstract value also contains the functions escape behaviour. Obviously, we have to find a counterpart to the notion of void augmentation. Essentially, we use the same approach as in Definition 3.10: For functional type, we require that the escape tag is zero and the functions result is void for all void arguments.

Definition 5.3 (Void Abstract Value)
An abstract value $a \in \mathsf{A}^t$ be is called *void abstract value* iff

- $a = \perp^t$ with $t \in S$.
- $a = (0, f)$ and $f(a_1, \ldots, a_n)$ is void for all void arguments $a_i \in \mathsf{A}^{t_i}$. ◁

Example: The value $\alpha((0, \mathsf{id})) = (0, \lambda(a).a) \in \mathsf{A}^{\mathtt{int}\to\mathtt{int}}$ is void. ◇

Lemma 5.3 (Properties of Void Abstract Value)

1. If $a \in \mathsf{A}^t$ is a void abstract value then all $a' \preceq^t a$ are void.
2. If $A \subseteq \mathsf{A}^t$ is a set of abstract values then all $a \in A$ are void iff $\bigsqcup^t A$ is void. □

Proof Simple induction on type t. q.e.d.

The next result show that the void abstract values can be used to identify values with void augmentation.

Lemma 5.4 (Void Abstract Values Approximate Void Augmentation)
If $\widehat{v} \in \widehat{\mathsf{CT}}^t$ such that $\alpha(\widehat{v})$ is a void abstract value then $\widehat{v}$ has void augmentation. □

Proof Induction on t

$t = bs \in BS$: Let $\widehat{v} = (b, v)$ such that $\alpha(\widehat{v})$ void
$\iff \perp^{bs} = \alpha(\widehat{v})$

$$
\begin{aligned}
&\iff 0 = \alpha((b,v)) \\
&\iff 0 = b \\
&\iff (0,v) = (b,v) \\
&\iff \oplus(v) = \widehat{v} \\
&\iff \widehat{v} \text{ has void augmentation}
\end{aligned}
$$

$t = cs \in CS$: $\alpha(\widehat{v})$ void

$$
\begin{aligned}
&\iff \alpha(\widehat{v}) = \bot^{cs} \\
&\iff \bigsqcup^{cs}\{\alpha_p(p) \mid p \in \widehat{v}\} = \bot^{cs} \\
&\iff \forall p \in \widehat{v} : \alpha_p(p) = \bot^{cs} \\
&\iff \forall p \in \widehat{v} : p = \bot^{cs} \text{ or } p \neq \bot^{cs} \text{ and } \alpha_p(p) = \bot^{cs} \\
&\iff \forall (b, c(w_1,\dots,w_n)) \in \widehat{v} : (b,(0,\dots,0)) \sqcup^{cs} \mathfrak{E}[\![c]\!](\alpha_1(w_1),\dots,\alpha_n(w_n)) = \bot^{cs} \\
&\qquad \text{where} \begin{cases} w_i \in \widehat{\mathsf{PT}}^{t_i},\ \alpha_i = \alpha_p & \text{if } t_i \in CS \\ w_i \in \widehat{\mathsf{CT}}^{t_i},\ \alpha_i = \alpha & \text{if } t_i \notin CS \end{cases} \quad 1 \le i \le n \\
&\iff \forall (b, c(w_1,\dots,w_n)) \in \widehat{v} : b = 0 \text{ and } \alpha_i(w_i) = \bot^{t_i} \\
&\qquad \text{where} \begin{cases} w_i \in \widehat{\mathsf{PT}}^{t_i},\ \alpha_i = \alpha_p & \text{if } t_i \in CS \\ w_i \in \widehat{\mathsf{CT}}^{t_i},\ \alpha_i = \alpha & \text{if } t_i \notin CS \end{cases} \quad 1 \le i \le n \\
&\overset{(*)}{\iff} \forall (b, c(w_1,\dots,w_n)) \in \widehat{v} : b = 0 \text{ and } \oplus_i(\ominus_i(w_i)) = w_i \\
&\qquad \text{where} \begin{cases} w_i \in \widehat{\mathsf{PT}}^{t_i},\ \oplus_i = \oplus_\bot,\ \ominus_i = \ominus_\bot & \text{if } t_i \in CS \\ w_i \in \widehat{\mathsf{CT}}^{t_i},\ \oplus_i = \oplus,\ \ominus_i = \ominus & \text{if } t_i \notin CS \end{cases} \quad 1 \le i \le n \\
&\iff \forall (b, c(w_1,\dots,w_n)) \in \widehat{v} : \oplus_\bot(\ominus_\bot((b, c(w_1,\dots,w_n)))) = (b, c(w_1,\dots,w_n)) \\
&\iff \forall p \in \widehat{v} : \oplus_\bot(\ominus_\bot(p)) = p \\
&\implies \forall p \in \widehat{v}\ \exists p' \in \widehat{v} : \oplus_\bot(\ominus_\bot(p)) = p' \text{ and } \forall p' \in \widehat{v}\ \exists p \in \widehat{v} : p' = \oplus_\bot(\ominus_\bot(p)) \\
&\iff \oplus(\ominus(\widehat{v})) = \oplus(\{\ominus_\bot(p) \mid p \in \widehat{v}\}) = \{\oplus_\bot(\ominus_\bot(p)) \mid p \in \widehat{v}\} = \widehat{v} \\
&\iff \widehat{v} \text{ has void augmentation}
\end{aligned}
$$

The equivalence $(*)$ can be shown by a trivial induction on the height of cs and the structure of p.

$t = t_1,\dots,t_m \to t_0$: Let $\widehat{v} = (b,f)$: $\alpha(\widehat{v})$ void

$$
\begin{aligned}
&\iff (b, \lambda(a_1,\dots,a_n).\bigsqcup^t\{\alpha(f(\widehat{v}_1,\dots,\widehat{v}_n)) \mid \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i},\ \alpha(\widehat{v}_i) \preceq^{t_i} a_i,\ 1 \le i \le n\}) \text{ void} \\
&\iff b = 0 \text{ and } \bigsqcup^{t_0}\{\alpha(f(\widehat{v}_1,\dots,\widehat{v}_n)) \mid \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i},\ \alpha(\widehat{v}_i) \preceq^{t_i} a_i,\ 1 \le i \le n\} \text{ void for all} \\
&\qquad \text{void } a_i \in \mathsf{A}^{t_i},\ 1 \le i \le n \\
&\iff b = 0 \text{ and } \alpha(f(\widehat{v}_1,\dots,\widehat{v}_n)) \text{ void for all } \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}, \alpha(\widehat{v}_i) \preceq^{t_i} a_i \text{ for all void } a_i \in \mathsf{A}^{t_i}, \\
&\qquad 1 \le i \le n \\
&\iff b = 0 \text{ and } f(\widehat{v}_1,\dots,\widehat{v}_n) \text{ has void augmentation for all } \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i}, \alpha(\widehat{v}_i) \preceq^{t_i} a_i \text{ for} \\
&\qquad \text{all void } a_i \in \mathsf{A}^{t_i},\ 1 \le i \le n \\
&\iff b = 0 \text{ and } f(\widehat{v}_1,\dots,\widehat{v}_n) \text{ has void augmentation for all } \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i} \text{ with void} \\
&\qquad \text{augmentation},\ 1 \le i \le n \\
&\iff \widehat{v} \text{ has void augmentation}
\end{aligned}
$$

q.e.d.

Unfortunately, we were not able to prove that void abstract values are an equivalent characterisation of values with void augmentation. However, we strongly believe that this is the case, since the only non–equivalence in the proof concerns partial terms and we were not able to construct a counterexample for the unproven implication.

Still, this lemma allows us to use $\mathfrak{E}$ as an indicator for non–escaping: If $\mathfrak{E}$ yields a void abstract value then we can be sure that the augmented semantics has void augmentation, i.e. nothing escapes.

Lemma 5.5 ($\mathfrak{E}$ Approximates Escaping)
For all programs P, expressions $e \in E^t$, environments $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$, and variables $x \in X$ holds that if $\mathfrak{E}[\![e]\!](\alpha(\widehat{\beta}), \mathfrak{E}[\![P]\!])$ is void then no parts of x escape from e. □

Proof Assume $\mathfrak{E}[\![e]\!](\widehat{\beta}, \mathfrak{E}[\![P]\!])$ is void. Lemma 5.1, Corollary 5.1, and Theorem 5.1 guarantee that $\alpha(\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\mathfrak{M}}[\![P]\!]))$ is void, too. We can conclude with Lemma 5.4. q.e.d.

5.4 Summary

In this chapter we proved that the escape analysis $\mathfrak{E}$ is a safe approximation of the augmented semantics $\widehat{\mathfrak{M}}$. We have characterised void abstract values which indicate void augmentation and used these results to give a means for testing non–escaping by using $\mathfrak{E}$.

Part II: Optimisations

6. The Semantics $\mathfrak{G}$: Modelling Eager Graph Reduction Denotationally

In this chapter, we introduce the operational model we use to implement the language F, to show the effect of escaping for this model, and to study the optimisations we can perform based on this knowledge.

The execution model is *graph reduction*. We chose this model, although it is well–known that eager graph reduction is not a complete but only correct implementation of the non–strict semantics. However, the memory behaviour of lazy graph reduction is much less predictable than those of eager graph reduction. Therefore, the optimisations we demonstrate in Chapter 7 can only be used for an eager operational model. In Section 8.5 we discuss how we can modify the optimisations to allow lazy evaluation.

Graph reduction was originally invented to implement the lambda calculus [Wad71]. In this original approach both program and data are represented in the graph. Nowadays, graph reduction is a synonym for *programmed graph reduction* where the program is not part of the graph, but operates on the data represented in the graph.

There are essentially two approaches for describing graph reduction: *abstract machines* and *graph rewrite systems*. Since our interest is focused on the removal of *garbage* in the graphs, the approach we choose must be able to express the handling of garbage.

Abstract machines like the *G–machine* [Aug87, Joh87], the *TIM* [FW87, Arg89], the *ABC machine* [PvE93], or the *STG* [PS89, Pey92] implement both *control structures* by transforming recursion to iteration and *data structures* by using graph reduction. They share the property that they "can be easily translated into any concrete machine code" [PL91a], and hence contain many details which are irrelevant for our interests. Especially the removal of recursion is a major obstacle, because a proof of correctness in this setting would also include the proof of correctness of the translation of recursion to iteration.

Graph rewrite systems [BvEG$^+$87, PvE93] are extensions of term rewrite systems which were introduced to include the notion of sharing. They are suited to describe lazy approaches, where a value is computed at most one time. However, there is no notion of garbage in graph rewrite systems.

Therefore, none of these approaches is suited for our needs. Instead, we introduce an additional level of abstraction to model graph reduction (see Figure 6.1). Since we are only interested in the representation of data structures and not in the implementation of control, it is convenient to avoid the additional obstacles imposed by an abstract machine. Instead, our "operational" model is a *denotational graph semantics* $\mathfrak{G}$, which uses an explicit graph component to represent data structures, but leaves the recursive structure of the programs unchanged. The graph component is unique, i.e. there is only one graph which is threaded through the evaluation. Therefore this approach can easily be mapped to an abstract machine where the graph is a global mutable component.

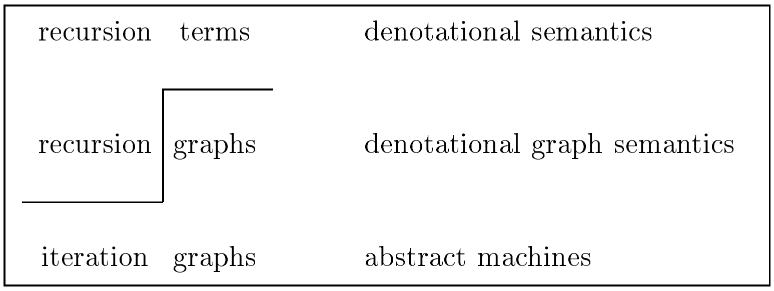


Fig. 6.1: Levels of Abstraction

Given a function definition $F(x_1, \ldots, x_n) := e$ with $x_i \in X^{t_i}$ and $e \in E^t$ the main idea is to associate a function of type $\mathsf{P}^{t_1} \times \cdots \times \mathsf{P}^{t_n} \times \mathsf{G} \to \mathsf{P}^t \times \mathsf{G}$ with this definition. The parameter of type G is the graph containing the arguments to the function. The i–th argument is represented by a pointer of type P^{t_i} into the graph. The result is a pair consisting of the modified graph and the pointer to the result in that graph.

The major task in defining $\mathfrak{G}$ is the definition of the domain for the graph component. The standard theory requires the semantic domains to be complete partially ordered sets. This allows the application of the Theorem of Knaster and Tarski and hence the definition of the semantics as least fixpoint.

In our context we cannot apply this theorem: The order on the graphs $\preccurlyeq$ must ignore the garbage, because otherwise removal of garbage would be a non–monotonic operation. In turn, two graphs $g_1 \neq g_2$ which differ only in the amount of garbage cannot be distinguished by this order, i.e. $g_1 \preccurlyeq g_2$ and $g_2 \preccurlyeq g_1$. But partially ordered sets require that the order is anti–symmetric and therefore we cannot model the graph domain as a cpo.

Instead we use a graph domain which is a *quasi ordered set*, i.e. a partially ordered set lacking anti-symmetry. There, neither the least element nor least fixpoints are uniquely determined, but only up to garbage. To allow the definition of the semantics, we develop an extension of the standard theory the standard theory that can handle such domains.

6.1 Heap Functions

We assume that we have an arbitrary but fixed family of *heap locations* $\mathsf{HL} = \langle \mathsf{HL}^t \mid t \in \mathsf{T}(S) \rangle$ where each HL^t is a set of typed heap locations.The graph is represented by a *heap function*, i.e. a storage mapping *heap locations (addresses)* to *heap nodes*. To allocate new heap cells, we have to know which address to choose next.

Definition 6.1 (Allocation Strategy)
An *allocation strategy* for HL is a family of functions $\mathsf{free} = \langle \mathsf{free}^t \mid t \in \mathsf{T}(S) \rangle$ such that for each $\mathsf{free}^t : \mathfrak{P}_{\text{fin}}(\mathsf{HL}^t) \to \mathsf{HL}^t$ and $T \subseteq \mathsf{HL}^t$, T finite, holds: $\mathsf{free}^t(T) \notin T$. $\lhd$

We can restrict the allocation strategy to finite subsets of P because all graph functions will have finite domains.

Example: The simplest example for locations and an allocation strategy is the set of natural numbers and "first–fit": $\mathsf{HL}^t := \mathbb{N}$, $\mathsf{free}^t(N) := \max\limits_N +1$. $\diamond$

For the remainder of the thesis we assume that we have a fixed allocation strategy.

Definition 6.2 (Heap Nodes, Heap Function)

1. We define the family of all *heap nodes* $\mathsf{HN} := \langle \mathsf{HN}^t \mid t \in \mathsf{T}(S)\rangle$, where the sets HN^t are defined as:

   (a) $\mathsf{HN}^{bs} := \{\langle v\rangle \mid v \in V^{bs}\}$ for all basic sort $t = bs \in BS$.

   (b) $\mathsf{HN}^{cs} := \{\langle c, l_1, \ldots, l_n\rangle \mid c \in C^{t_1,\ldots,t_n \to cs}, n \in \mathbb{N}, l_i \in \mathsf{HL}^{t_i}, 1 \leq i \leq n\}$ for all constructed sorts $t = cs \in CS$.

   (c) $\mathsf{HN}^{t_1,\ldots,t_n \to t_e} := \{\langle \varphi\rangle \mid \varphi \in (\Omega \cup DF)^{t_1,\ldots,t_n\to t_e}\}$
   $\cup\, \{\langle @, l_f, l_1, \ldots, l_m\rangle \mid l_f \in \mathsf{HL}^{t'_1,\ldots,t'_m,t_1,\ldots,t_n \to t_e}, l_i \in \mathsf{HL}^{t'_i}, 1 \leq i \leq m\}$
   for all functional types $t = t_1, \ldots, t_n \to t_e \in \mathsf{T}(S)$.

2. A family of functions $h = \langle h^t \mid t \in \mathsf{T}(S)\rangle$ with $h^t : \mathsf{HL}^t \dashrightarrow \mathsf{HN}^t$ is called *heap function* iff $\mathsf{Dom}(h^t)$ is finite for all $t \in \mathsf{T}(S)$. We write: $h : \mathsf{HL} \dashrightarrow \mathsf{HN}$. $\lhd$

Corresponding to the types, we have four different kinds of heap nodes:

1. Heap nodes containing basic values; this *boxed representation* [Pey87] can be inefficient in realistic implementations but is easier to deal with in this context.

2. Heap nodes containing a constructor and locations for its arguments.

3. Heap nodes containing a function name (defined or intrinsic); these nodes represent functions without arguments. In abstract machines, the functions names are replaced by pointers to the code of the function.

4. Heap nodes containing locations for a function node and for already supplied arguments for that function; these nodes represent partially applied functions and are called *closures*.

Example: In Figure 6.2 we give an example for a heap function. Each box in the figure represents one graph node, the corresponding locations are given in the upper right corner of the box. The graph contains one shared node l_4. Assume that we have $\mathtt{append} \in DF^{\mathtt{ListOfInt},\mathtt{ListOfInt}\to\mathtt{ListOfInt}}$. Using the above formalisation, the graph consists of four functions: $h^{\mathtt{int}}$, $h^{\mathtt{ListOfInt}}$, $h^{\mathtt{ListOfInt}\to\mathtt{ListOfInt}}$, and $h^{\mathtt{ListOfInt},\mathtt{ListOfInt}\to\mathtt{ListOfInt}}$ with domains $\mathsf{Dom}(h^{\mathtt{int}}) = \{l_4\}$, $\mathsf{Dom}(h^{\mathtt{ListOfInt}}) = \{l_3, l_5, l_6, l_7\}$, $\mathsf{Dom}(h^{\mathtt{ListOfInt}\to\mathtt{ListOfInt}}) = \{l_1\}$, and $\mathsf{Dom}(h^{\mathtt{ListOfInt},\mathtt{ListOfInt}\to\mathtt{ListOfInt}}) = \{l_2\}$. $\diamond$

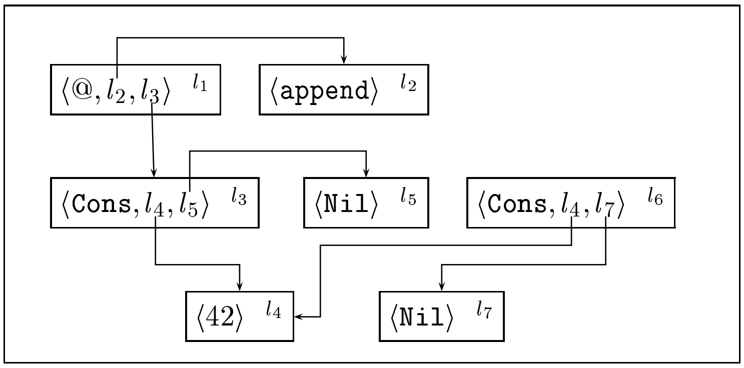


Fig. 6.2: Demo Graph g_{demo}

Note that there exist various modifications of this approach, like "unboxed values as first class citizens" [PL91b] or a "spineless" representation of partially applied function [PS89, Pey92]. However, all these are variations of the same basic approach and can easily be studied in this setting.

Modelling lazy graph reduction would only require the addition of a fifth kind of heap nodes: *thunks* are essentially the same as closures except that they represent applications which are saturated but not (yet) evaluated.

We can identify one special heap function, the *empty heap* $g_\emptyset$ with $\mathsf{Dom}(g_\emptyset^t) = \emptyset$ for all $t \in \mathsf{T}(S)$.

Not all heap functions are representations of a graph and not all graphs are reasonable in the context of this thesis:

- A heap h may have *dangling pointers*, i.e. there exists a heap cell such that the locations stored in the heap cell are invalid (not in $\mathsf{Dom}(h)$).
- A heap may by cyclic.

To formalise this, we need the notion of dependence.

Definition 6.3 (Dependant Locations)
Let $h : \mathsf{HL} \dashrightarrow \mathsf{HN}$ be a heap function, $l \in \mathsf{HL}^t$, $l' \in \mathsf{HL}^{t'}$. The location l *depends on* the location l' iff $l \in \mathsf{Dom}(h^t)$ and either $h(l) = \langle c, l_1, \ldots, l_n \rangle$ or $h(l) = \langle @, l_1, l_2, \ldots, l_n \rangle$ and $l' = l_j$ for one $1 \leq j \leq n$.

We write $l \rightarrowtail l'$. As usual, $\rightarrowtail^+$ denotes the transitive closure, and $\rightarrowtail^*$ denotes the transitive and reflexive closure of $\rightarrowtail$. For $l \in \mathsf{HL}^t$, $\overrightarrow{l} := \{l' \in \mathsf{HL}^{t'} \mid l \rightarrowtail^* l'\}$ denotes the locations on which l depends, and for $L \subseteq \mathsf{HL}$, $\overrightarrow{L} := \bigcup_{l \in L} \overrightarrow{l}$ denotes the locations on which any of L depends. ◁

In terms of graphs theory we can characterise $\overrightarrow{l}$ as the graph component spanned by l.

Example: For the graph function g_{demo} from Figure 6.2 we have $\overrightarrow{l_1} = \{l_1, l_2, l_3, l_4, l_5\}$ and $\overrightarrow{l_6} = \{l_6, l_4, l_7\}$. ◇

Definition 6.4 (Graph Function)
Let $h : \mathsf{HL} \dashrightarrow \mathsf{HN}$ be a heap function. We call h a *graph function* iff

1. h has no dangling pointers, i.e. for all $l \in \mathsf{HL}$ holds that $\overrightarrow{l} \subseteq \mathsf{Dom}(h)$
2. h is not cyclic, i.e. for all $l \in \mathsf{Dom}(h)$ holds: $l \not\rightarrowtail^{+} l$ $\triangleleft$

Note that the empty heap function $g_\emptyset$ is a graph function, *the empty graph.*

To relate heaps with the domains of the denotational semantics $\widehat{\mathfrak{M}}$, we define the following function.

Definition 6.5 (Representation Function rep)
Let $g : \mathsf{HL} \dashrightarrow \mathsf{HN}$, $g \neq g_\emptyset$ be a non–empty graph function. The *representation function* $\mathsf{rep}_g : \mathsf{Dom}(g^t) \times \mathsf{Env}(DF, \mathsf{Ops}) \to \mathsf{CT}^t$ is defined by induction on t:

$t = bs \in BS :$ $\mathsf{rep}_g(l, \sigma) := v$ if $g(l) = \langle v \rangle$.

$t = cs \in CS :$ $\mathsf{rep}_g(l, \sigma) := c(v_1, \ldots, v_n)$ if $g(l) = \langle c, l_1, \ldots, l_n \rangle$ and $v_i = \mathsf{rep}_g(l_i, \sigma)$.

$t = t_1, \ldots, t_n \to t_0 \in \mathsf{T}(S)$:

$$\mathsf{rep}_g(l, \sigma) := \begin{cases} \mathfrak{M}[\![\varphi]\!]([], \sigma) & \text{if } g(l) = \langle \varphi \rangle,\ \varphi \in (\Omega \cup DF)^{t_1, \ldots, t_n \to t_e} \\[2ex] \lambda(v_{m+1}, \ldots, v_n).f(v_1, \ldots, v_m, v_{m+1}, \ldots, v_n) & \\ & \text{if } g(l) = \langle @, l_f, l_1, \ldots, l_m \rangle,\ f = \mathsf{rep}_g(l_f, \sigma), \\ & \quad v_i = \mathsf{rep}_g(l_i, \sigma),\ 1 \leq i \leq n \end{cases} \triangleleft$$

Remarks:

- The properties from Definition 6.4 guarantee that rep_g is well–defined for all graph functions g.
- Since the graphs contain nodes with the *names* of functions, we need an environment for defined functions to assign values to partial applications.
- Different graph functions may represent the same value caused by sharing and distribution of the nodes.
- No terms containing $\perp$ can be result of rep_g; the possibility of non–terminating computations is handled differently here.
- No infinite values i.e. infinite ideals can be result of rep_g.
- Hence, the function rep_g is neither injective nor surjective.

Example: To give some examples for rep, we use the graph function g_{demo} given in Figure 6.2. With $\sigma = [\mathsf{append}/app]$ we obtain the following representations:

1. $\mathsf{rep}_g(l_4, \sigma) = 42$

2. $\mathsf{rep}_g(l_5, \sigma) = \mathsf{rep}_g(l_7, \sigma) = \mathtt{Nil}$

3. $\mathsf{rep}_g(l_3, \sigma) = \mathsf{rep}_g(l_6, \sigma) = \mathtt{Cons}(42, \mathtt{Nil})$

4. $\mathsf{rep}_g(l_2, \sigma) = \mathit{app}$

5. $\mathsf{rep}_g(l_1, \sigma) = \lambda(l).\mathit{app}(\mathtt{Cons}(42, \mathtt{Nil}), l)$ ◇

6.2 *The Graph Domain* G

The removal of *garbage* in a graph is the aim of the applications we consider. Hence we must formalise the notion of garbage, which requires that we can determine which heap nodes are active. By adding such information to a graph function we obtain our representation of a graph. In an implementation of graph reduction with an abstract machine, this information can be obtained by inspecting the call stack.

Furthermore, we introduce an order on the set of graphs which we need to model functions on graphs denotationally.

Definition 6.6 (Graph Domain G)
The graph domain $\langle \mathsf{G}, \preccurlyeq \rangle$ is defined as:

1. $\mathsf{G} := \{(g, L) \mid g : \mathsf{HL} \dashrightarrow \mathsf{HN} \text{ graph function}, L \subseteq \mathsf{Dom}(g)\}$

2. $G_1 \preccurlyeq G_2$ iff $G_i = (g_i, L_i) \in \mathsf{G}$ $(i = 1, 2)$ with $L_1 \subseteq L_2$ and $\forall l_1 \in L_1$: $g_1(l_1) = g_2(l_1)$.

3. for $G = (g, L) \in \mathsf{G}$, L is the set of *active locations*, and $\mathsf{Dom}(g) \setminus L$ is the set of *garbage locations*. ◁

Example: If we choose the graph function g_{demo} from Figure 6.2 and $L = \{l_1, \ldots, l_5\}$ then the spine consisting of $\{l_6, l_7\}$ is garbage. ◇

Remarks:

- A graph $G \in \mathsf{G}$ is a tuple consisting of a graph function and a set of active locations.

- The active locations define which parts of the graph functions are relevant. All other graph locations are *garbage*. The order $\preccurlyeq$ is the usual graph inclusion restricted to the set of active locations.

- Our notion of garbage is more general than the usual one. Classically, garbage cells are those which are not reachable from any active location. But since we made no restriction on the second component of a graph, it may occur in this setting that garbage cells are reachable from active cells and vice versa. This is useful, because it may be the case that a cell is reachable from an active location, but is never going to be used again. Our approach allows to declare such a location as garbage. For instance, *region inference* [TT94, BTV96] considers such garbage cells.

- Because the order $\preccurlyeq$ ignores the values of the graph functions at garbage locations two graphs which differ only at those locations cannot be distinguished by $\preccurlyeq$. Hence, $\preccurlyeq$ is not anti–symmetric and consequently it is *not a partially ordered set*. Consider $(g_i, \emptyset) \in \mathsf{G}$ for $i = 1, 2$ such that $g_1 \neq g_2$. Since there are no active locations, we have $g_1 \preccurlyeq g_2$ and vice versa.

Corollary 6.1
The graph domain G is a quasi ordered set (qos). □

Because the standard theory of denotational semantics requires complete partially ordered sets as denotational domains, we have to extend this theory accordingly. Essentially, our aim is to use the fixpoint characterisation from Tarski's Theorem. In Section 6.4 we show how to obtain a similar possibility in this broader framework.

We need one additional definition before we can introduce the semantics $\mathfrak{G}$. To model partial functions on locations and graphs we need representations for undefined locations.

Definition 6.7 (Pointers)
For all $t \in \mathsf{T}(S)$ we define the *pointer domain* $\langle \mathsf{P}^t,\ \trianglelefteq^t \rangle$ by

$$\mathsf{P}^t := \mathsf{HL}^t \cup \{\mathsf{NULL}^t\} \qquad \mathsf{NULL}^t \trianglelefteq^t p,\ p \trianglelefteq^t p \quad \forall p \in \mathsf{P}^t$$

The family P of pointer domains is defined as $\mathsf{P} := \langle \mathsf{P}^t \mid t \in \mathsf{T}(S) \rangle$. ◁

We obtain the pointer domains by adding additional elements to the sets of locations. The resulting structures are flat, and hence the following corollary is obvious.

Corollary 6.2
The pointer domains P are flat cpos. □

6.3 Graph Expression Semantics

We define the graph expression semantics $\mathfrak{G}$ such that the representation of all data is done in the graph component. Therefore, we introduce a new notion which allows this allocation of a new node in a graph.

Definition 6.8 (Graph Allocation)
Let $G = (g, L) \in \mathsf{G}$ be a graph and let $hn \in \mathsf{HN}^t$ be a graph node. The pair consisting of location and graph $G + hn \in \mathsf{HL}^t \times \mathsf{G}$ which results from allocation of the node hn in G is defined as $G + hn := (l', G')$ where $l' = \mathsf{free}^t(\mathsf{Dom}(g))$, and $G' = (g[l'/hn], L \cup \{l'\})$ if the resulting function is a heap function. ◁

Example: The allocation of an additional constructor node in the demo graph from Figure 6.2 $(g_{\text{demo}}, \mathsf{Dom}(g_{\text{demo}})) + \langle \mathtt{Cons}, l_4, l_6 \rangle$ yields the graph in Figure 6.3. ◇

We start the definition of the graph semantics by defining the graph semantics of intrinsic functions.

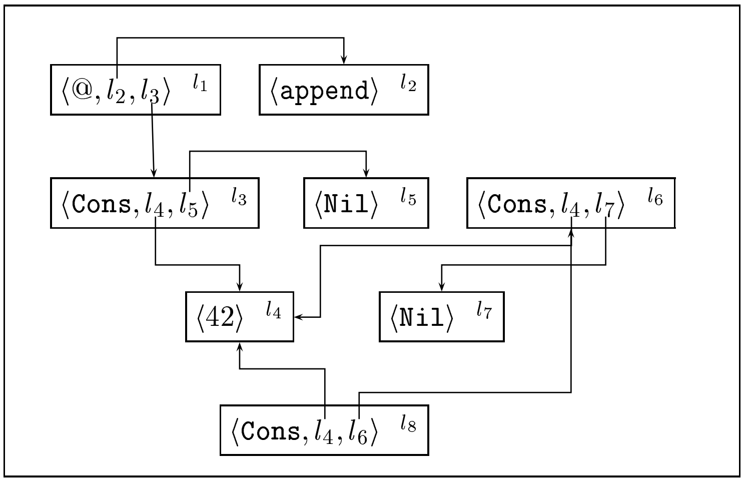


Fig. 6.3: $(g_{\text{demo}}, \mathsf{Dom}(g_{\text{demo}})) + \langle \mathtt{Cons}, l_4, l_6\rangle$

Definition 6.9 (Graph Semantics of Intrinsic Functions $\mathfrak{G}$)
Let $f \in \Omega^{t_1,\dots,t_n \to t}$ be an intrinsic function. The *graph semantics of f* is defined as a function $\mathfrak{G}[\![f]\!] : \mathsf{P}^{t_1} \times \cdots \times \mathsf{P}^{t_n} \times \mathsf{G} \to \mathsf{P}^{t} \times \mathsf{G}$

1. $\mathfrak{G}[\![bf]\!](p_1,\dots,p_n,(g,L)) := \begin{cases} (\mathsf{NULL}^t, (g_\emptyset, \emptyset)) & \text{if } \exists 1 \le j \le n \text{ with } p_j \notin \mathsf{Dom}(g) \\ (g,L) + \langle \mathfrak{M}[\![bf]\!](g(p_1),\dots,g(p_n))\rangle & \text{otherwise} \end{cases}$

   for all $bf \in BF$

2. $\mathfrak{G}[\![c]\!](p_1,\dots,p_n,(g,L)) := (g,L) + \langle c, g(p_1),\dots,g(p_n)\rangle$ for all constructors $c \in C$

3. $\mathfrak{G}[\![\mathtt{sel}\overset{j}{-}c]\!](p,(g,L)) := \begin{cases} (p_j, (g,L)) & \text{if } p \in \mathsf{Dom}(g),\ g(p) = \langle c, l_1,\dots,l_m\rangle,\ 1 \le j \le m \\ (\mathsf{NULL}^{t_j}, (g_\emptyset, \emptyset)) & \text{otherwise} \end{cases}$

   for all selectors $\mathtt{sel}\overset{j}{-}c \in CSel$

4. $\mathfrak{G}[\![\mathtt{is}-c]\!](p,(g,L)) := \begin{cases} (\mathsf{NULL}^{\mathtt{bool}}, (g_\emptyset, \emptyset)) & \text{if } p \notin \mathsf{Dom}(g) \\ (g,L) + \langle \mathtt{true}\rangle & \text{if } p \in \mathsf{Dom}(g),\ g(p) = \langle c, l_1,\dots,l_m\rangle \\ (g,L) + \langle \mathtt{false}\rangle & \text{otherwise} \end{cases}$

   for all $\mathtt{is}-c \in CTest$ ◁

Remarks:

- For basic functions, the graph semantics unpacks the arguments, applies the function, and packs the result in a new heap cell.
- For constructors, a new constructor cell is created, which contains the references to the arguments. Here, sharing of graph components can occur.
- For constructor tests, the graph is inspected whether the location points to a cell for the constructor. The boolean result is stored in a newly created heap cell.
- For selectors, the corresponding entry in the constructor node is returned. This is the only case where the graph is not modified.

- If one of the arguments is an invalid pointer then the result will also contain an invalid pointer. This is especially the case if any of the argument pointers is equal to NULL. Hence the semantics is strict.

The denotational semantics of an expression is determined by the interpretation of variables X, defined functions DF, the semantics of intrinsic functions, and, of course, a graph. The interpretation of a defined functions is an operation on the graph domain.

DEFINITION 6.10 (GRAPH OPERATIONS)
The family of *graph operations* is defined as $\mathsf{GOps} := \langle \mathsf{GOps}^{t_1,\ldots,t_n\to t} \mid t_1,\ldots,t_n \to t \in \mathsf{T}(S)\rangle$, where $\mathsf{GOps}^{t_1,\ldots,t_n\to t}$ is defined as $\mathsf{GOps}^{t_1,\ldots,t_n\to t} := \{f : \mathsf{P}^{t_1} \times \cdots \times \mathsf{P}^{t_n} \times \mathsf{G} \to \mathsf{P}^{t} \times \mathsf{G}\}$. $\lhd$

Functions are represented by graph nodes: intrinsic functions or user–defined functions by simple nodes and functions resulting from partial applications by application nodes (closures). Hence, to execute saturated applications, we need an auxiliary function which collects all arguments while following a linked chain of closures. This functions uses the semantics of intrinsic functions and an environment for user–defined functions.

DEFINITION 6.11 (EXECUTION FUNCTION exec)
For all functional types $t_1,\ldots,t_n \to t \in \mathsf{T}(S)$ we simultaneously define *the execution function* $\mathsf{exec}^{t_1,\ldots,t_n\to t} : \mathsf{Env}(DF,\mathsf{GOps}) \times \mathsf{P}^{t_1,\ldots,t_n\to t} \times \mathsf{G} \to \mathsf{GOps}^{t_1,\ldots,t_n\to t}$ in the following way. for $\theta \in \mathsf{Env}(DF,\mathsf{GOps})$, $p_f \in \mathsf{P}^{t_1,\ldots,t_n\to t}$, and $(g,L) \in \mathsf{G}$:

- $\mathsf{exec}^{t_1,\ldots,t_n\to t}(\theta,p_f,(g,L)) := \lambda(p_1,\ldots,p_n,L).(\mathsf{NULL}^t,(g_\emptyset,\emptyset))$ if $p_f \notin \mathsf{Dom}(g)$.
- $\mathsf{exec}^{t_1,\ldots,t_n\to t}(\theta,p_f,(g,L)) := \mathfrak{G}[\![f]\!]$ if $g(p_f) = \langle f\rangle$, $f \in \Omega^{t_1,\ldots,t_n\to t}$.
- $\mathsf{exec}^{t_1,\ldots,t_n\to t}(\theta,p_f,(g,L)) := \theta(F)$ if $g(p_f) = \langle F\rangle$, $F \in DF^{t_1,\ldots,t_n\to t}$.
- $\mathsf{exec}^{t_1,\ldots,t_n\to t}(\theta,p_f,(g,L)) := \lambda(p_1,\ldots,p_n,G).f(p'_1,\ldots,p'_m,p_1,\ldots,p_n,G)$ if $g(p_f) = \langle @,p'_f,p'_1,\ldots,p'_m\rangle$, $p'_f \in \mathsf{P}^{t'_1,\ldots,t'_m,t_1,\ldots,t_n\to t}$, and $f = \mathsf{exec}^{t'_1,\ldots,t'_m,t_1,\ldots,t_n\to t}(\theta,p'_f,(g,L))$. $\lhd$

Remarks:

- Although exec is not defined by induction on the type structure, it is well–defined because the graph is finite and not cyclic.
- Application of exec to an environment, a pointer, and a graph yields a graph operation as result, i.e. a function which takes pointers and a graph as arguments.

Example: With an environment θ, the graph function g_{demo} from Figure 6.2, and some set L we obtain

$$\begin{aligned}
&(\mathsf{exec}^{\mathtt{ListOfInt}\to\mathtt{ListOfInt}}(\theta,l_1,(g,L)))(l_6,(g,L))\\
&= (\mathsf{exec}^{\mathtt{ListOfInt},\mathtt{ListOfInt}\to\mathtt{ListOfInt}}(\theta,l_2,(g,L)))(l_3,l_6,(g,L))\\
&= (\theta(\mathsf{append}))(l_3,l_6,(g,L)) \qquad \diamond
\end{aligned}$$

Before we can define the graph expression semantics, we need one more auxiliary function.

Definition 6.12 (Garbage Creation Function mkgarb)
The *garbage creation function* $\mathsf{mkgarb} : \mathfrak{P}(\mathsf{HL}) \times (\mathsf{P}^t \times \mathsf{G}) \to \mathsf{P}^t \times \mathsf{G}$ is defined as

$$\mathsf{mkgarb}(L', (p, (g, L))) := \begin{cases} (\mathsf{NULL}^t, (g_\emptyset, \emptyset)) & \text{if } p = \mathsf{NULL}^t \\ (p, (g, L' \cup \overrightarrow{p})) & \text{otherwise} \end{cases} \qquad \lhd$$

The purpose of this function is to identify heap cells as garbage by omitting their locations in the set of active locations. Every location which was in L but is neither in L' nor in $\overrightarrow{p}$ becomes garbage.

Definition 6.13 (Graph Expression Semantics $\mathfrak{G}$)
Let $e \in E^t$ be an expression, $G \in \mathsf{G}$ be a graph, $\xi \in \mathsf{Env}(X, \mathsf{P})$ be an environment for variables, and $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ be an environment for defined functions. The *graph semantics of* e $\mathfrak{G}[\![e]\!](\xi, \theta, G) \in \mathsf{P}^t \times \mathsf{G}$ is defined inductively on the structure of e:

- $\mathfrak{G}[\![x]\!](\xi, \theta, G) := (\xi(x), G)$ for $x \in X^t$
- $\mathfrak{G}[\![F]\!](\xi, \theta, G) := G + \langle F \rangle$ for $F \in DF^t$
- $\mathfrak{G}[\![f]\!](\xi, \theta, G) := G + \langle f \rangle$ for $f \in \Omega^t$
- $\mathfrak{G}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\xi, \theta, (g, L)) := \mathsf{mkgarb}(L, (p_r, G_r))$ with

  $$(p_r, G_r) := \begin{cases} f(p_1, \ldots, p_m, (g_m, L_m)) \text{ if } m = n,\ f := \mathsf{exec}^{t_1, \ldots, t_m \to t}(\theta, p_0, (g_m, L_m)) \\ (g_m, L_m) + \langle @, p_0, p_1, \ldots, p_m \rangle \text{ if } 1 \leq m < n \end{cases}$$

  where $(p_0, (g_0, L_0)) := \mathfrak{G}[\![e_0]\!](\xi, \theta, (g, L))$
  $(p_j, (g_j, L_j)) := \mathfrak{G}[\![e_j]\!](\xi, \theta, (g_{j-1}, L_{j-1}))$ for $1 \leq j \leq m$
  for $e_0 \in E^{t_1, \ldots, t_n \to t_0}$ and $e_i \in E^{t_i}$ $(1 \leq i \leq m)$
- $\mathfrak{G}[\![\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2]\!](\xi, \theta, (g, L)) :=$

  $$\begin{cases} (\mathsf{NULL}^t, (g_\emptyset, \emptyset)) & \text{if } p' \notin \mathsf{Dom}(g') \\ \mathsf{mkgarb}(L, \mathfrak{G}[\![e_1]\!](\xi, \theta, (g', L'))) & \text{if } p' \in \mathsf{Dom}(g'),\ g'(p') = \langle \texttt{true} \rangle \\ \mathsf{mkgarb}(L, \mathfrak{G}[\![e_2]\!](\xi, \theta, (g', L'))) & \text{if } p' \in \mathsf{Dom}(g'),\ g'(p') = \langle \texttt{false} \rangle \end{cases}$$

  where $(p', (g', L')) := \mathfrak{G}[\![e_0]\!](\xi, \theta, G)$ for $e_0 \in E^{\mathsf{bool}}$ and $e_1, e_2 \in E^t$ $\lhd$

Remarks:

- The graph is an unique parameter in the semantics. It always has an unique state, which is threaded through the evaluation. Therefore this semantics can easily be implemented by an abstract machine where the graph is a global mutable component.
- In contrast to the semantics $\mathfrak{M}$ and $\widehat{\mathfrak{M}}$, this expression semantics is eager; if one argument pointer is NULL it is not element of $\mathsf{Dom}(g)$ and hence the resulting pointer is also NULL.
- The evaluation order is fixed to be *left–to–right*.
- We can determine those constructs where garbage can occur:

In applications: Graph cells which were allocated during the evaluation of arguments of an application, but are neither shared nor part of the result.

In conditionals: Graph cells which were allocated during the evaluation of the conditional expression.

Examples:

1. Our first example is the expression $e = \texttt{(Cons x l)}$ with the graph (g, L) with $g = g_{\text{demo}}$ from Figure 6.2, $L = \mathsf{Dom}(g_{\text{demo}})$, and the variable environment $\xi = [\texttt{x}/l_4, \texttt{l}/l_3]$.

$$
\begin{aligned}
&\mathfrak{G}[\![\texttt{(Cons x l)}]\!](\xi, \theta, (g, L)) \\
&\quad = \mathsf{mkgarb}(L, \mathsf{exec}^{\texttt{int},\texttt{ListOfInt}\to\texttt{ListOfInt}}(\theta, p_0, (g_2, L_2))(p_1, p_2, (g_2, L_2))) \\
&\qquad \text{where } (p_0, (g_0, L_0)) = \mathfrak{G}[\![\texttt{Cons}]\!](\xi, \theta, (g, L)) \\
&\qquad\qquad = (g, L) + \langle\texttt{Cons}\rangle \\
&\qquad\qquad = (l_8, (g[l_8/\langle\texttt{Cons}\rangle], L \cup \{l_8\})) \\
&\qquad (p_1, (g_1, L_1)) = \mathfrak{G}[\![\texttt{x}]\!](\xi, \theta, (g_0, L_0)) \\
&\qquad\qquad = (\xi(\texttt{x}), (g_0, L_0)) \\
&\qquad\qquad = (l_4, (g_0, L_0)) \\
&\qquad (p_2, (g_2, L_2)) = \mathfrak{G}[\![\texttt{l}]\!](\xi, \theta, (g_1, L_1)) \\
&\qquad\qquad = (\xi(\texttt{l}), (g_0, L_0)) \\
&\qquad\qquad = (l_3, (g_0, L_0)) \\
&\quad = \mathsf{mkgarb}(L, \mathfrak{G}[\![\texttt{Cons}]\!](l_8, l_4, l_3, g[l_8/\langle\texttt{Cons}\rangle], L \cup \{l_8\})) \\
&\quad = \mathsf{mkgarb}(L, (g[l_8/\langle\texttt{Cons}\rangle], L \cup \{l_8\}) + \langle\texttt{Cons}, l_4, l_3\rangle) \\
&\quad = \mathsf{mkgarb}(L, (l_9, (g[l_8/\langle\texttt{Cons}\rangle, l_9/\langle\texttt{Cons}, l_4, l_3\rangle], L \cup \{l_8, l_9\}))) \\
&\quad = (l_9, (g[l_8/\langle\texttt{Cons}\rangle, l_9/\langle\texttt{Cons}, l_4, l_3\rangle], L \cup \overrightarrow{l_9})) \\
&\quad = (l_9, (g[l_8/\langle\texttt{Cons}\rangle, l_9/\langle\texttt{Cons}, l_4, l_3\rangle], L \cup \{l_9\}))
\end{aligned}
$$

One cell of garbage is created: The node $\langle\texttt{Cons}\rangle$ at location l_8 is not part of the result.

2. The second example is the right–hand side of the usual `append` definition:

$$
\begin{aligned}
e = {} & \texttt{if (is-Nil l) then Nil} \\
& \qquad \texttt{else (Cons (sel}^1\texttt{-Cons l) (append (sel}^2\texttt{-Cons l) m))}
\end{aligned}
$$

Like in the previous example, we use the graph function $g = g_{\text{demo}}$ from Figure 6.2 and $L = \mathsf{Dom}(g)$. Furthermore, we use environments $\xi = [\texttt{l}/l_3, \texttt{m}/l_7]$ and $\theta = [\texttt{append}/app^1]$ with $app^1 : \mathsf{P}^{\texttt{ListOfInt}} \times \mathsf{P}^{\texttt{ListOfInt}} \times \mathsf{G} \to \mathsf{P}^{\texttt{ListOfInt}} \times \mathsf{G}$ defined as

$$
app^1(l_1, l_2, (g, L)) = \begin{cases} l_2 & \text{if } g(l_1) = \langle\texttt{Nil}\rangle \\ (\mathsf{NULL}^{\texttt{ListOfInt}}, (g_\emptyset, \emptyset)) & \text{otherwise} \end{cases}
$$

This function implements appending to an empty list.

$$
\begin{aligned}
&\mathfrak{G}[\![e]\!](\xi, \theta, (g, L)) \\
&\quad = \begin{cases} (\mathsf{NULL}^t, (g_\emptyset, \emptyset)) & \text{if } p' \notin \mathsf{Dom}(g') \\ \mathsf{mkgarb}(L, \mathfrak{G}[\![e_1]\!](\xi, \theta, (g', L'))) & \text{if } p' \in \mathsf{Dom}(g'),\ g'(p') = \langle\texttt{true}\rangle \\ \mathsf{mkgarb}(L, \mathfrak{G}[\![e_2]\!](\xi, \theta, (g', L'))) & \text{if } p' \in \mathsf{Dom}(g'),\ g'(p') = \langle\texttt{false}\rangle \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\quad \text{where } (p', (g', L')) &&= \mathfrak{G}[\![\texttt{(is−Nil l)}]\!](\xi, \theta, (g, L)) \\
& &&= (l_9, (g[l_8/\langle\texttt{is−Nil}\rangle, l_9/\langle\texttt{true}\rangle], L \cup \{l_9\})) \\
&= \mathsf{mkgarb}(L, \mathfrak{G}[\![\texttt{(Cons ...)}]\!](\xi, \theta, (g', L'))) \\
&= \mathsf{mkgarb}(L, \mathsf{mkgarb}(L, \mathsf{exec}(\theta, p_0, (g_2, L_2))(p_1, p_2, (g_2, L_2)))) \\
&\quad \text{where } (p_0, (g_0, L_0)) &&= \mathfrak{G}[\![\texttt{Cons}]\!](\xi, \theta, (g', L')) \\
& &&= (l_{10}, (g'[l_{10}/\langle\texttt{Cons}\rangle], L' \cup \{l_{10}\})) \\
&\quad (p_1, (g_1, L_1)) &&= \mathfrak{G}[\![\texttt{(sel}^1\texttt{−Cons l)}]\!](\xi, \theta, (g_0, L_0)) \\
& &&= (l_4, (g_0[l_{11}/\langle\texttt{sel}^1\texttt{−Cons}\rangle], L_0)) \\
&\quad (p_2, (g_2, L_2)) &&= \mathfrak{G}[\![\texttt{(append (sel}^2\texttt{−Cons l) m))}]\!](\xi, \theta, (g_1, L_1)) \\
& &&= (l_7, (g_1[l_{12}/\langle\texttt{append}\rangle, l_{13}/\langle\texttt{sel}^2\texttt{−Cons}\rangle], L_0)) \\
&= \mathsf{mkgarb}(L, \mathsf{mkgarb}(L, (l_{14}, (g_2[l_{14}/\langle\texttt{Cons}, l_4, l_7\rangle], L_2 \cup \{l_{14}\})))) \\
&= \mathsf{mkgarb}(L, (l_{14}, (g_2[l_{14}/\langle\texttt{Cons}, l_4, l_7\rangle], L \cup \overleftrightarrow{l_{14}}))) \\
&= (l_{14}, (g_2[l_{14}/\langle\texttt{Cons}, l_4, l_7\rangle], L \cup \{l_{14}\}))
\end{aligned}
$$

The resulting graph $(g_2[l_{14}/\langle\texttt{Cons}, l_4, l_7\rangle], L \cup \{l_{14}\})$ is shown in Figure 6.4. It contains six garbage cells, which are set with a dashed border.

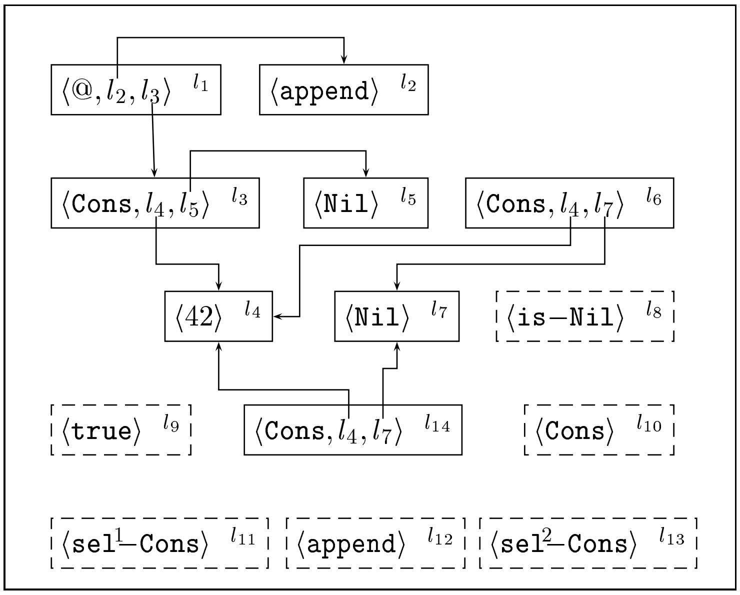


Fig. 6.4: Resulting Graph of Example 2

As before, we use the expression semantics to associate a transformation with a program.

Definition 6.14 (Graph Semantic Transformation)
Given a program $P = (F(x_{j1}, \ldots, x_{jn_j}) := e_j \mid 1 \le j \le p)$ with $F_j \in DF^{t_{j1}, \ldots, t_{jn_j} \to t_j}$, $x_{ji} \in X^{t_{ji}}$, and $e_j \in E^{t_j}$ with variables $\{x_{j1}, \ldots, x_{jn_j}\}$ for $1 \le j \le p$ the *graph semantic transformation for* P, $\Phi_{\mathfrak{G},P} : \mathsf{FS}_{\mathfrak{G},P} \to \mathsf{FS}_{\mathfrak{G},P}$ on the function space $\mathsf{FS}_{\mathfrak{G},P} := \prod_{j=1}^{p} \mathsf{GOps}^{t_{j1}, \ldots, t_{jn_j} \to t_j}$

is defined as

$$\Phi_{\mathfrak{G},P}(g_1,\ldots,g_p) \quad := \quad \begin{pmatrix} \lambda(p_{11},\ldots,p_{1n_1},G_1).\mathfrak{G}[\![e_1]\!]([x_{11}/p_{11},\ldots,x_{1n_1}/p_{1n_1}],\theta,G_1) \\ \vdots \\ \lambda(p_{p1},\ldots,p_{pn_p},G_p).\mathfrak{G}[\![e_p]\!]([x_{p1}/p_{p1},\ldots,x_{pn_p}/p_{pn_p}],\theta,G_p) \end{pmatrix}$$
$$\text{where } \theta := [F_1/g_1,\ldots,F_p/g_p] \qquad \triangleleft$$

Example: For the `append` program $P_{\texttt{append}}$, we have the following transformation

$$\Phi_{\mathfrak{G},P_{\texttt{append}}}(g) = \lambda(p_1,p_2,G).\mathfrak{G}[\![\texttt{if (is-Nil l) ...}]\!]([\texttt{l}/p_1,\texttt{l'}/p_2],[\texttt{append}/g],G) \qquad \diamond$$

But unlike before, the fixpoint theorem of Knaster and Tarski does *not* guarantee that the least fixpoint of $\Phi_{\mathfrak{G},P}$ exists and can be represented as least upper bound of successive applications of $\Phi_{\mathfrak{G},P}$. The graph domain G is not a cpo. It is not even a pos, but only a qos. Consequently the function space $\mathsf{FS}_{\mathfrak{G},P}$ is not a cpo either.

Hence, we must generalise the fixpoint theorem to deal with this more general setting. The following problems arise when we try to do so:

1. Because the graph order $\preccurlyeq$ is not symmetric the least element of a set $F \subseteq \mathsf{FS}_{\mathfrak{G},P}$ (if it exists) is not determined uniquely. Consequently, we do not have an unique least element of $\mathsf{FS}_{\mathfrak{G},P}$.

2. Even worse, G is not complete: Consider the infinite chain of (finite) graphs $G_i \in \mathsf{G}$ with $G_0 = (g_\emptyset,\emptyset)$ and $G_{i+1} = G_i + \langle\texttt{true}\rangle$ for $i \in \mathbb{N}$. Obviously, we have $G_i \preccurlyeq G_{i+1}$ but a least upper bound of the chain $\{G_i \mid i \in \mathbb{N}\}$ is an infinite graph $(g_\infty, \mathsf{HL}^{\texttt{bool}})$ with $g_\infty(l) = \langle\texttt{true}\rangle$ for all $l \in \mathsf{HL}^{\texttt{bool}}$, which hence cannot be an element of G.

3. But even if we choose one least element $\bot_{\mathsf{FS}_{\mathfrak{G},P}}$ to be the starting point of the chain of successive iterations and can ensure that the chain has least upper bounds, the limit $\bigsqcup\{\Phi^i_{\mathfrak{G},P}(\bot_{\mathsf{FS}_{\mathfrak{G},P}}) \mid i \in \mathbb{N}\}$ is not unique either.

4. In addition, when we try to choose one element of the limit as semantics, we must ensure that it is indeed a fixpoint of $\Phi_{\mathfrak{G},P}$. Not all elements must have this property, because although they are identical in the parts which can be distinguished by the order, they may indeed differ in the garbage parts.

We can generalise the standard theory by using the following observations:

1. It does not matter which least element is the starting point of the chain of successive applications of the transformation.

2. Although the least upper bound is not unique in quasi ordered sets, we can identify a notion of convergence, such that the limit fulfils the properties we need.

We achieve this aim by examining the proof of the fixpoint theorem of Knaster and Tarski (Theorem A.1). We can observe that

- It does not actually need the completeness of the whole cpo. In fact, it just needs the chain of successive applications of the transformation to have a least upper bound.
- It is independent of the structure of the cpos involved; especially, it does not need the cpos to be sets of functions.

In the next section, we use this observations to formalise a generalisation of the standard fixpoint theory.

6.4 Fixpoints in Quasi Ordered Sets

The usual approach for the handling of recursion in a denotational context is based on cpos and the fixpoint theorem by Knaster and Tarski. Another approach, typically used in the context of concurrency ([dBZ82]), is based on metric spaces and the fixpoint theorem by Banach (see [BMC94, MCB96] for a more detailed discussion of the metric space approach). We use techniques and notions from both approaches to obtain results for qos.

We start by generalising some basic notions of partially ordered sets. We then can demonstrate their deficiencies in the context of qos.

Definition 6.15 (Basic Notions)
Let $\langle Q, \preccurlyeq\rangle, \langle Q', \preccurlyeq'\rangle$ be qos.

1. The *set of least elements of a set* $X \subseteq Q$ is defined as
$$\mathsf{least}(X) := \{y \in Q \mid y \preccurlyeq x \text{ for every } x \in X\}$$
The *set of least elements of* Q is defined as $\mathsf{least}(Q)$.
2. A non–empty set $D \subseteq Q$ is *directed* iff $\forall x, y \in D\ \exists z \in D$ such that $x \preccurlyeq z$ and $y \preccurlyeq z$.
3. The *set of least upper bounds of a directed set* $D \subseteq Q$ is defined as
$$\mathsf{lub}(D) := \mathsf{least}(\{y \in Q \mid x \preccurlyeq y \text{ for all } x \in D\})$$
4. A function $f : Q \to Q'$ is called *monotonic* iff for all $x, y \in Q$ with $x \preccurlyeq y$ holds that
$$f(x) \preccurlyeq' f(y) \qquad \lhd$$

Here we can see the major problems: Neither least element nor least upper bound are determined uniquely. This implies that for a given function f, even if we can guarantee that $x \in \mathsf{lub}(\{f^i(b) \mid i \in \mathbb{N}\})$ exists (for some $b \in \mathsf{least}(D)$), it is not uniquely determined either. Hence, we need a way of selecting an unique element. However, if we try to transfer the notion of continuity of a function f with the condition

$$f(\mathsf{lub}(D)) = \mathsf{lub}(f(D))$$

we have a condition between *sets*, and not between *elements*. Consequently, even if we can select an $x \in \mathsf{lub}(\{f^i(b) \mid i \in \mathbb{N}\})$ for an f which fulfils the above condition, we can only conclude that $f(x) \in \mathsf{lub}(\{f^i(b) \mid i \in \mathbb{N}\})$ but *it does not need to be a fixpoint of* f.

Example: We use the following structure as a running example: $\mathbf{Q} := \langle (\mathbb{N} \cup \{\perp\}) \times \mathbb{N},\ \preccurlyeq_{\mathbf{Q}} \rangle$, where $(x, y) \preccurlyeq_{\mathbf{Q}} (x', y')$ iff $x =\perp$ or $x = x'$. Obviously, $\mathbf{Q}$ is a qos and the least elements of $\mathbf{Q}$ are the elements of the set $\{(\perp, n) \mid n \in \mathbb{N}\}$. $\mathbf{Q}$ can be seen as augmentation of the flat cpo $\mathbb{N} \cup \{\perp\}$ with a second component of natural numbers. ◇

We show that an element of the least upper bound does not need to be a least element after the introduction of product and function space qos.

Like in pos, monotonic functions preserve directed sets.

Corollary 6.3
Let $\langle Q,\ \preccurlyeq \rangle, \langle Q',\ \preccurlyeq' \rangle$ be qos, $f : Q \to Q'$ be a monotonic function, and $D \subseteq Q$ be a directed set. Then $f(D)$ is directed. □

The next lemma shows that qos have the same closure properties wrt. product and function spaces as pos.

Lemma 6.1 (Closure Properties)
Let $\langle Q_1,\ \preccurlyeq_1 \rangle$, $\langle Q_2,\ \preccurlyeq_2 \rangle$ be qos.

1. the *product structure* $\langle Q_1 \times Q_2,\ \preccurlyeq_{Q_1 \times Q_2} \rangle$, where the relation $\preccurlyeq_{Q_1 \times Q_2}$ is defined as

$$(x, y) \preccurlyeq_{Q_1 \times Q_2} (x', y') \ :\Longleftrightarrow\ x \preccurlyeq_1 x', y \preccurlyeq_2 y'$$

is a qos.

2. the *function space* $\langle [Q_1 \to Q_2],\ \preccurlyeq_{Q_1 \to Q_2} \rangle$, where the relation $\preccurlyeq_{Q_1 \to Q_2}$ is defined as

$$f \preccurlyeq_{Q_1 \to Q_2} g \ :\Longleftrightarrow\ f(x) \preccurlyeq_2 g(x) \text{ for all } x \in Q_1$$

is a qos. □

We now focus on function space qos, since we need to exploit their special structure to obtain our main result. As already mentioned, using the lub is not sufficient to obtain a single value as limit, because lub is a set of values. However, in function space qos, we can identify functions which have a unique limit within lub.

Example: Consider for instance our running example $\mathbf{Q}$ and the directed set of functions $\{(I_n)_{n \in \mathbb{N}}\}$, with $I_n \in [\mathbf{Q} \to \mathbf{Q}]$ defined by

$$I_n((x, y)) = \begin{cases} (x, y) & \text{if } x \in \mathbb{N} \text{ and } x < n \\ (\perp, y) & \text{otherwise} \end{cases} \qquad \forall n \in \mathbb{N}$$

For each $n \in \mathbb{N}$, I_n is the identity on the segment $\{(\perp, y), (0, y), \ldots, (n, y)\}$ for all $y \in \mathbb{N}$. For the remaining arguments, $I_n((x, y))$ is equal to $(\perp, y)$.
The least upper bound $\mathsf{lub}(\{(I_n)_{n \in \mathbb{N}}\})$ is the set of functions which are the identity on the first component, i.e.

$$\mathsf{lub}(\{(I_n)_{n \in \mathbb{N}}\}) = \{I : \mathbf{Q} \to \mathbf{Q} \mid I((x, y)) = (x, z) \text{ for all } (x, y) \in \mathbf{Q} \text{ and some } z \in \mathbb{N}\}$$

Here, the behaviour on the second component is completely ignored, since $\preccurlyeq_{\mathbf{Q}}$ ignores the second component. However, intuitively the sequence $(I_n)_{n\in\mathbb{N}}$ uniquely determines a single function in $\mathsf{lub}(\{(I_n)_{n\in\mathbb{N}}\})$, the identity function id. $\diamond$

We introduce a notion of convergence, which captures this intuition.

Definition 6.16 (Convergent Sequence of Functions)
Let A and B be sets. A sequence of functions $(f_n)_{n\in\mathbb{N}}$, $f_n : A \to B$ is called *convergent to* $f : A \to B$ ($\lim\limits_{n\to\infty} f_n = f$) iff for all $x \in A$ exists $i \in \mathbb{N}$ such that $f(x) = f_i(x) = f_{i+j}(x)$ for all $j \in \mathbb{N}$. $\lhd$

Example: The example functions $(I_n)_{n\in\mathbb{N}}$ are convergent: For $(\bot, y)$ we have index 0, and for (x, y) we have index $x + 1$. The limit is $\lim\limits_{n\to\infty} I_n = \mathsf{id}$. $\diamond$

Note that the sets A and B do not need to have any structure. Furthermore, this notion of convergence is related to the convergence notion for metric spaces ([SHLG94]). To show that, we assume $A = \mathbb{N}$, since in semantics countable sets are typically used. On the set of functions $[\mathbb{N} \to B] := \{f \mid f : \mathbb{N} \to B\}$ we define

$$\begin{aligned} \varrho : [\mathbb{N} \to B] &\times [\mathbb{N} \to B] \to \mathbb{R}^{\geq 0} \\ \varrho(f, g) \quad := &\sum_{\substack{n\in\mathbb{N} \\ f(n)\neq g(n)}} 2^{-n} \end{aligned}$$

It uses the geometric series $\sum_{n=0}^{\infty} a_1 q^n$, which converges to $\frac{a_1}{1-q}$. In our case we have $a_1 = 1$ and $q = \frac{1}{2}$, which means that $\varrho(f, g) = 2$ for functions with $f(n) \neq g(n)$ for all $n \in \mathbb{N}$.

The function ϱ is a *metric*, because it fulfils the following conditions: (1) $\varrho(f, g) = 0$ iff $f = g$, (2) $\varrho(f, g) = \varrho(g, f)$, and (3) $\varrho(f, h) \leq \varrho(f, g) + \varrho(g, h)$ for all $f, g, h \in [\mathbb{N} \to B]$. In metric spaces we have the notion of convergence, which is defined in the following way for this metric: A sequence of functions $(f_n)_{n\in\mathbb{N}}$ converges to f in the metric space $([\mathbb{N} \to B], \varrho)$ iff for all $\varepsilon > 0$ exists $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ holds: $\varrho(f, f_n) < \varepsilon$.

It is easy to see that $(f_n)_{n\in\mathbb{N}}$ converges to f corresponding to this notion iff it converges to f in the sense of Definition 6.16.

When we consider transformations, i.e. functions on function spaces, we get the result that all transformations preserve convergence.

Lemma 6.2
Let A, B be sets and $F := \{f \mid f : A \to B\}$ be the set of functions from A to B, $\Phi : F \to F$ be a transformation, and $(f_n)_{n\in\mathbb{N}} \subseteq F$ a convergent sequence. Then we have that $(\Phi(f_n))_{n\in\mathbb{N}} \subseteq F$ is a convergent sequence. $\square$

Proof Let $(f_n)_{n\in\mathbb{N}}$ be convergent, i.e. for all $x \in A$ exists $i \in \mathbb{N}$ such that $f_i(x) = f_{i+j}(x)$ for all $j \in \mathbb{N}$. Hence, $\Phi(f_i)(x) = \Phi(f_{i+j})(x)$ for all $j \in \mathbb{N}$ and consequently $(\Phi(f_n))_{n\in\mathbb{N}}$ is convergent. q.e.d.

However, we need more. To obtain a fixpoint, we need the possibility to exchange application of the transformation and limit, i.e. we need functions which preserve the limit.

Definition 6.17 (Continuous Function)
Let A, B be sets and $F := \{f \mid f : A \to B\}$ be the set of functions. A transformation $\Phi : F \to F$ is called *continuous* iff for all convergent sequences $(f_n)_{n\in\mathbb{N}} \subseteq F$ holds:

$$\Phi(\lim_{n\to\infty} f_n) = \lim_{n\to\infty} \Phi(f_n) \qquad \triangleleft$$

Again, this is an instance of the notion of continuity in metric spaces. For instance, if $A = \mathbb{N}$ then a transformation F is continuous iff it is continuous in our special metric space, i.e. for all $f_0 \in \{f \mid f : \mathbb{N} \to B\}$ and $\varepsilon > 0$ exists $\delta > 0$ such that $\varrho(f, f_0) < \delta$ implies $\varrho(\Phi(f), \Phi(f_0)) < \varepsilon$.

The next result establishes a connection between this notion of limit and the limit in the usual sense for qos.

Lemma 6.3
Let $\langle [Q_1 \to Q_2], \preccurlyeq_{Q_1\to Q_2}\rangle$ be a function space qos like in Lemma 6.1 and $(f_n)_{n\in\mathbb{N}} \subseteq [Q_1 \to Q_2]$ be a convergent sequence of functions, such that $f_i \preccurlyeq_{Q_1\to Q_2} f_{i+1}$ for all $i \in \mathbb{N}$. Then we have:

$$\lim_{n\to\infty} f_n \in \mathsf{lub}(\{(f_n)_{n\in\mathbb{N}}\}) \qquad \square$$

Proof We know that the least upper bound is defined, because $\{(f_n)_{n\in\mathbb{N}}\}$ is a directed set. Let $f := \lim_{n\to\infty} f_n$. We first prove that f is an upper bound for all elements of the sequence, i.e. for all $n \in \mathbb{N}$ holds

$$f_n \preccurlyeq_{Q_1\to Q_2} f$$

By definition of $\preccurlyeq_{Q_1\to Q_2}$ this is equivalent to $f_n(x) \preccurlyeq_2 f(x)$ for all $x \in Q_1$. Let $x \in Q_1$ be fixed, and i be the first index such that $f_i(x) = f_{i+j}(x)$ for all $j \in \mathbb{N}$. Either $n < i$ then we have $f_n(x) \preccurlyeq_2 f_{n+1}(x) \preccurlyeq_2 \cdots \preccurlyeq_2 f_i(x) = f(x)$, or else $f_n(x) = f(x)$.

It remains to be shown that f is a least element of the upper bounds of $(f_n)_{n\in\mathbb{N}}$. Let g be a least upper bound, i.e. $g \in \mathsf{lub}(\{(f_n)_{n\in\mathbb{N}}\})$. We have to prove that

$$f \preccurlyeq_{Q_1\to Q_2} g$$

i.e. that for all $x \in Q_1$ holds: $f(x) \preccurlyeq_2 g(x)$. Assume that there exists $x \in Q_1$ such that $f(x) \not\preccurlyeq_2 g(x)$. By definition of f there exists $i \in \mathbb{N}$ such that $g(x) = f_i(x)$ and hence $f_i(x) \not\preccurlyeq_2 g(x)$ which is a contradiction to $g \in \mathsf{lub}(\{(f_n)_{n\in\mathbb{N}}\})$. q.e.d.

With these preparations we can formulate the main result of this section.

Theorem 6.1 (Fixpoint Theorem for Function Space Quasi Ordered Sets)
Let $Q = \langle [Q_1 \to Q_2], \preccurlyeq_{Q_1\to Q_2}\rangle$ be a function space qos and $\Phi : Q \to Q$ be a monotonic and continuous transformation. For all least elements $f_b \in \mathsf{least}(Q)$ holds that if $(\Phi^n(f_b))_{n\in\mathbb{N}}$ is convergent then $\lim_{n\to\infty} \Phi^n(f_b)$ is a least fixpoint of Φ. $\square$

PROOF We know that $f_\infty := \lim_{n\to\infty} \Phi^n(f_b)$ exists because by assumption $(\Phi^n(f_b))_{n\in\mathbb{N}}$ is convergent. We have to show that f_∞ is fixpoint of Φ:

$$\begin{aligned}\Phi(f_\infty) &= \Phi(\lim_{n\to\infty} \Phi^n(f_b)) \\ &= \lim_{n\to\infty} \Phi(\Phi^n(f_b)) \\ &= \lim_{n\to\infty} \Phi^n(f_b) \\ &= f_\infty\end{aligned}$$

What remains is the proof that f_∞ is least among the set of fixpoints of Φ. Let g be another fixpoint of Φ. By definition of $\preccurlyeq_{Q_1\to Q_2}$ we have

$$\begin{aligned} & f_\infty \preccurlyeq_{Q_1\to Q_2} g \\ \iff\quad & f_\infty(x) \preccurlyeq_2 g(x) \quad \forall x \in Q_1\end{aligned}$$

Let x be fixed. Because f_∞ is a limit there exists an $i \in \mathbb{N}$ such that $f_\infty(x) = (\Phi^i(f_b))(x)$. Given that, we have to show that

$$(\Phi^i(f_b))(x) \preccurlyeq_2 g(x)$$

We show that this is true for all $0 \le j \le i$ by induction on $j \in \mathbb{N}$:

$j = 0$: We have $(\Phi^0(f_b))(x) = f_b(x) \preccurlyeq_2 g(x)$ because $f_b \in \mathsf{least}(Q)$.

$j \to j+1$: By induction hypothesis we have $(\Phi^j(f_b))(x) \preccurlyeq_2 g(x)$. Because Φ is monotonic, this implies $(\Phi^{j+1}(f_b))(x) \preccurlyeq_2 (\Phi(g))(x) = g(x)$. q.e.d.

Example: For the qos $\mathbf{Q}$, we define a transformation $\Phi : [\mathbf{Q} \to \mathbf{Q}] \to [\mathbf{Q} \to \mathbf{Q}]$ by

$$\Phi(I) := \lambda(x,y).\begin{cases}(\bot, y) & \text{if } x = \bot \\ (0, y) & \text{if } x = 0 \\ (x'+1, y') & \text{if } x \notin \{\bot, 0\} \text{ with } (x', y') = I((x-1, y))\end{cases}$$

It is easy to show that Φ is continuous. We choose a least element $B \in \mathsf{least}([\mathbf{Q} \to \mathbf{Q}])$ by $B((x,y)) = (\bot, y)$. We then have

$$\Phi^n(B) = I_n \qquad \forall n \in \mathbb{N}$$

which means that we can apply the preceding theorem and obtain the identity function as limit. $\diamond$

To use our new fixpoint theorem for defining a graph program semantics, we need one small extension to deal with products of function spaces.

DEFINITION 6.18 (COMPONENTWISE CONVERGENT SEQUENCE, LIMIT)
Let $Q = \langle Q_1 \times \cdots \times Q_m,\ \preccurlyeq_{Q_1\times\cdots\times Q_m}\rangle$ be a product qos, where $Q_i = \langle [D_i \to R_i],\ \preccurlyeq_{D_i\to R_i}\rangle$ is a function space qos for all $1 \le i \le m$. A sequence of tuples of functions $((f_{1,n}, \ldots, f_{m,n}))_{n\in\mathbb{N}}$, $f_{j,n} \in Q_j$ is called *componentwise convergent* iff for all $1 \le j \le n$ $(f_{j,n})_{n\in\mathbb{N}}$ is convergent. The *limit* of $((f_{1,n}, \ldots, f_{m,n}))_{n\in\mathbb{N}}$ is the tuple of functions

$$\lim_{n\to\infty}(f_{1,n}, \ldots, f_{m,n}) := (\lim_{n\to\infty} f_{1,n}, \ldots, \lim_{n\to\infty} f_{m,n}) \qquad \triangleleft$$

Definition 6.19 (Continuous Transformation)
Let $Q = \langle Q_1 \times \cdots \times Q_m,\ \preccurlyeq_{Q_1\times\cdots\times Q_m}\rangle$ be a product qos, where $Q_i = \langle [D_i \to R_i],\ \preccurlyeq_{D_i\to R_i}\rangle$ is a function space qos for all $1 \leq i \leq m$. A transformation $\Phi : Q \to Q$ is *continuous* iff for all convergent sequences $((f_{1,n},\ldots,f_{m,n}))_{n\in\mathbb{N}}$ holds:

$$\Phi(\lim_{n\to\infty}(f_{1,n},\ldots,f_{m,n})) = \lim_{n\to\infty}\Phi((f_{1,n},\ldots,f_{m,n}))) \qquad \lhd$$

Theorem 6.2 (Fixpoint Theorem for Product of Function Spaces QOS)
Let $Q = \langle Q_1 \times \cdots \times Q_m,\ \preccurlyeq_{Q_1\times\cdots\times Q_m}\rangle$ be a product qos, where $Q_i = \langle [D_i \to R_i],\ \preccurlyeq_{D_i\to R_i}\rangle$ is a function space qos for all $1 \leq i \leq m$ and $\Phi : Q \to Q$ is a monotonic and continuous transformation. For all least elements $t_b \in \mathsf{least}(Q)$ holds that if $(\Phi^n(t_b))_{n\in\mathbb{N}}$ is componentwise convergent then $\lim\limits_{n\to\infty}\Phi^n(t_b)$ is a least fixpoint of Φ. $\square$

Before we start to use the generalisation of the fixpoint theorem, we discuss the relation to the standard theory. Here we can use two different points of view:

1. Consider the special case where the qos are also pos.
2. Use the well–known fact [Wec92] that every qos induces an equivalence relation and hence a pos, and use the standard theory for the latter.

For the special case that the qos is a pos, the induced pos is isomorphic to the pos, hence we solely consider the second approach.

Definition 6.20 (Equivalence Relation induced by Quasi Ordered Set)
Let $\langle Q,\ \preccurlyeq\rangle$ be a qos. The *equivalence relation induced by* $\preccurlyeq$ is defined as

$$x \equiv_{\preccurlyeq} y \ :\Longleftrightarrow\ x \preccurlyeq y \text{ and } y \preccurlyeq x \qquad \text{for all } x, y \in Q \qquad \lhd$$

The relation $\equiv_{\preccurlyeq}$ is symmetric by definition and transitive and reflexive because $\preccurlyeq$ is a quasi ordered set.

Corollary 6.4 (Partial Ordered Set induced by Quasi Ordered Set)
Let $\langle Q,\ \preccurlyeq\rangle$ be a qos. The structure defined as $\langle Q/_{\equiv_{\preccurlyeq}},\ \preccurlyeq/_{\equiv_{\preccurlyeq}}\rangle$ is a pos, *the partially ordered set induced by* $\langle Q,\ \preccurlyeq\rangle$. $\square$

Of course, one approach to find fixpoints in qos would be to consider the induced pos instead. This would yield an equivalence class of solutions in the qos without the choice to designate a single element as solution. All additional structure in the qos would be ignored, which is not satisfactory. However, we can use the induced pos as a reference point. This is of interest especially in the context of proving the correctness of an annotated semantics wrt. a standard semantics. All which is to be done there is to prove that the induced pos of the annotated semantics is isomorphic to the standard pos.

The next result shows that our notions for qos are conservative extensions of the well–known notions for pos.

Corollary 6.5
Let $\langle Q,\ \preccurlyeq\rangle$ be a qos.

1. With $X \subseteq Q$ we have

$$\mathsf{least}(X) = \begin{cases} L & \text{if } L \text{ is the least element of } X/_{\equiv_{\preccurlyeq}} \text{ in } \langle Q/_{\equiv_{\preccurlyeq}},\ \preccurlyeq/_{\equiv_{\preccurlyeq}}\rangle \\ \emptyset & \text{if the least element does not exist} \end{cases}$$

2. A set $D \subseteq Q$ is a directed set iff $D/_{\equiv_{\preccurlyeq}}$ is a directed set in $\langle Q/_{\equiv_{\preccurlyeq}},\ \preccurlyeq/_{\equiv_{\preccurlyeq}}\rangle$.

3. Let $D \subseteq Q$ be a directed set:

$$\mathsf{lub}(D) = \begin{cases} \bigsqcup D/_{\equiv_{\preccurlyeq}} & \text{if it exists in } \langle Q/_{\equiv_{\preccurlyeq}},\ \preccurlyeq/_{\equiv_{\preccurlyeq}}\rangle \\ \emptyset & \text{otherwise} \end{cases}$$ □

With this result and the remark after Corollary 6.4 we know that for qos which are pos the limit is identical to the least upper bound of the partial order. Hence, we obtain the same result for function space pos with continuous transformations and convergent sequence of approximations. What remains to be investigated is how the different conditions "continuous transformations" and "convergent sequence of approximations" relate.

6.5 Graph Program Semantics

To use Theorem 6.2, we have to check its conditions.

Lemma 6.4

1. The structure $\langle \mathsf{P}^t \times \mathsf{G},\ \preccurlyeq_{\mathsf{P}^t\times\mathsf{G}}\rangle$ is a product space qos for all types $t \in \mathsf{T}(S)$.

2. The structure $\langle \mathsf{GOps}^{t_1,\ldots,t_n\to t},\ \preccurlyeq_{\to\mathsf{P}^t\times\mathsf{G}}\rangle$, with $\preccurlyeq_{\to\mathsf{P}^t\times\mathsf{G}}$ defined as

$$f \preccurlyeq_{\to\mathsf{P}^t\times\mathsf{G}} g \iff f(p_1,\ldots,p_n,G) \preccurlyeq_{\mathsf{P}^t\times\mathsf{G}} g(p_1,\ldots,p_n,G)$$

   for all $p_i \in \mathsf{P}^{t_i}$, $1 \le i \le n$, $G \in \mathsf{G}$ is a function space qos for all types $t_1,\ldots,t_n \to t \in \mathsf{T}(S)$.

3. The structure $\langle \mathsf{FS}_{\mathfrak{G},P},\ \preccurlyeq_{\mathfrak{G},P}\rangle$, with $\preccurlyeq_{\mathfrak{G},P}$ defined as

$$(f_1,\ldots,f_p) \preccurlyeq_{\mathfrak{G},P} (g_1,\ldots,g_p) \iff f_i \preccurlyeq_{\to\mathsf{P}^t\times\mathsf{G}} g_i \quad \forall 1 \le i \le p$$

   is a product of function space qos for all programs P with p definitions, i.e. $P = (F(x_{j1},\ldots,x_{jn_j}) := e_j \mid 1 \le j \le p)$ with $F_j \in DF^{t_{j1},\ldots,t_{jn_j}\to t_j}$, $x_{ji} \in X^{t_{ji}}$, and $e_j \in E^{t_j}$ with variables $\{x_{j1},\ldots,x_{jn_j}\}$ for $1 \le j \le p$.

4. The transformation $\Phi_{\mathfrak{G},P}$ is a monotonic transformation on $\langle \mathsf{FS}_{\mathfrak{G},P},\ \preccurlyeq_{\mathfrak{G},P}\rangle$ for all programs P. □

Now we have to select a least element in $\langle \mathsf{FS}_{\mathfrak{G},P},\ \preccurlyeq_{\mathfrak{G},P}\rangle$. Here we choose

$$T_b := (\lambda(p_{11},\ldots,p_{1n_1},G_1).(\mathsf{NULL}^{t_1},(g_\emptyset,\emptyset)),\ldots,\lambda(p_{p1},\ldots,p_{pn_p},G_p).(\mathsf{NULL}^{t_p},(g_\emptyset,\emptyset)))$$

the tuple of functions assigning each argument the least element in $\preccurlyeq_{\mathsf{P}^t\times\mathsf{G}}$. The only actual choice we have here is how much garbage the initial graphs contain. However, it is sensible to use the graph without garbage.

Obviously, $\Phi_{\mathfrak{G},P}$ is monotonic. We omit the proof that $\Phi_{\mathfrak{G},P}$ is continuous.

It remains to be checked that the sequence

$$(\Phi^n_{\mathfrak{G},P}(T_b))_{n\in\mathbb{N}}$$

is componentwise convergent. Our next aim is to show a stronger result; if we consider the j–th component of the i–th element of the sequence $(\Phi^i_{\mathfrak{G},P}(T_b))_j$, then if $(\Phi^i_{\mathfrak{G},P}(T_b))_j = (T)_j \neq (T_b)_j$ then we have also $(\Phi^{i+k}_{\mathfrak{G},P}(T_b))_j = (T)_j$ for all k. This means that as soon as the sequence has reached a defined result for one function of a program, than this result is fixed. To formalise this result, we need a new notation.

Definition 6.21 (Stable Value, Stable Function)
Let $(p,G),(p',G')\in\mathsf{P}^t\times\mathsf{G}$. The value (p',G') is called *stable wrt.* (p,G) iff either $(p,G) = (\mathsf{NULL}^t,(g_\emptyset,\emptyset))$ of $(p,G)=(p',G')$.

Let $f,f'\in\mathsf{GOps}^{t_1,\ldots,t_n\to t}$ be graph operations. The function f' is called *stable wrt. f'* iff $f'(p_1,\ldots,p_n,G)$ is stable wrt. $f(p_1,\ldots,p_n,G)$ for all $G\in\mathsf{G}$ and $p_i\in\mathsf{P}^{t_i}$ $(1\le i\le n)$. $\lhd$

The following result shows how the auxiliary functions used by the expression semantics respect the above notion.

Corollary 6.6 (Graph Allocation Respects Stability)
Let $(p,G),(p',G')\in\mathsf{P}^t\times\mathsf{G}$ such that (p',G') is stable wrt. (p,G), and let $hn\in\mathsf{HN}^t$ be a graph node. Then we have that $G+hn$ is stable wrt. $G'+hn$. $\square$

Corollary 6.7 (mkgarb Respects Stability)
Let $(p,G),(p',G')\in\mathsf{P}^t\times\mathsf{G}$ such that (p',G') is stable wrt. (p,G), and let $L\subseteq\mathsf{HL}$. Then we have that $\mathsf{mkgarb}(L',(p',G'))$ is stable wrt. $\mathsf{mkgarb}(L',(p,G))$. $\square$

Lemma 6.5 (exec Respects Stability)
Let $\theta,\theta'\in\mathsf{Env}(DF,\mathsf{GOps})$ such that $\theta'(F)$ is stable wrt. $\theta(F)$ for all $F\in DF^{t'_1,\ldots,t'_n\to t'}$. Furthermore, let $G,G'\in\mathsf{G}$ be graphs, $p_f,p'_f\in\mathsf{P}^{t_1,\ldots,t_n\to t}$ be pointers such that (p'_f,G') is stable wrt. (p_f,G), and $p_i,p'_i\in\mathsf{P}^{t_i}$ be pointers such that (p'_i,G') is stable wrt. (p_i,G) for $1\le i\le n$. Then we have that $\mathsf{exec}^{t_1,\ldots,t_n\to t}(\theta,p'_f,G')(p'_1,\ldots,p'_n,G')$ is stable wrt. $\mathsf{exec}^{t_1,\ldots,t_n\to t}(\theta,p_f,G)(p_1,\ldots,p_n,G)$. $\square$

Proof Let $G=(g,L)$, $G'=(g',L')$. If

$$\mathsf{exec}^{t_1,\ldots,t_n\to t}(\theta,p_f,(g,L))(p_1,\ldots,p_n,(g,L)) = (\mathsf{NULL}^t,(g_\emptyset,\emptyset))$$

then we have nothing to show. Therefore we assume that

$$\mathsf{exec}^{t_1,\ldots,t_n\to t}(\theta, p_f, (g,L))(p_1,\ldots,p_n,(g,L)) = (p_r,(g_r,L_r)) \neq (\mathsf{NULL}^t,(g_\emptyset,\emptyset))$$

We distinguish four cases:

1. If $p_f \notin \mathsf{Dom}(g)$, then we have $(p_r,(g_r,L_r)) = (\mathsf{NULL}^t,(g_\emptyset,\emptyset))$, which cannot happen.

2. If $g(p_f) = \langle f\rangle$, $f \in \Omega^{t_1,\ldots,t_n\to t}$, then we have $(p_r,(g_r,L_r)) = \mathfrak{G}[\![f]\!](p_1,\ldots,p_n,(g,L))$. Assume that for any $1 \le i \le n$ holds that $(p_i,G) \neq (p'_i,G)$. Then we know that $(p_i,G) = (\mathsf{NULL}^t,(g_\emptyset,\emptyset))$ and because $\mathfrak{G}[\![f]\!]$ is strict we can conclude that $(p_r,(g_r,L_r))$ is equal to $(\mathsf{NULL}^t,(g_\emptyset,\emptyset))$, which cannot happen. Hence, $(p_i,G) = (p'_i,G)$ for all $1 \le i \le n$ and consequently $\mathfrak{G}[\![f]\!](p'_1,\ldots,p'_n,(g',L')) = (p_r,(g_r,L_r))$.

3. If $g(p_f) = \langle F\rangle$, $F \in DF^{t_1,\ldots,t_n\to t}$ a defined function, then we know that $(p_r,(g_r,L_r)) = (\theta(F))(p_1,\ldots,p_n,(g,L))$. Given that, the result immediately follows from the assumption that $\theta'(F)$ is stable wrt. $\theta(F)$ and the same argument as in the preceding case for the identity of the arguments.

4. Otherwise we have $g(p_f) = \langle @, l_f, l_1,\ldots,l_m\rangle$. Because the graph is finite and not cyclic we can reduce this case to one of the previous cases in finitely many steps. q.e.d.

LEMMA 6.6
Let $e \in E^t$ be an expression, $G \in \mathsf{G}$ be a graph, $\xi \in \mathsf{Env}(X,\mathsf{P})$ be an environment for variables, and $\theta,\theta' \in \mathsf{Env}(DF,\mathsf{GOps})$ be environments for defined functions. If $\theta'(F)$ is stable wrt. $\theta(F)$ for all $F \in DF^{t'_1,\ldots,t'_n\to t'}$ then $\mathfrak{G}[\![e]\!](\xi,\theta',G)$ is stable wrt. $\mathfrak{G}[\![e]\!](\xi,\theta,G)$. □

PROOF If $\mathfrak{G}[\![e]\!](\xi,\theta,G) = (\mathsf{NULL}^t,(g_\emptyset,\emptyset))$ then we have nothing to show. Therefore we assume that $\mathfrak{G}[\![e]\!](\xi,\theta,G) = (p,G) \neq (\mathsf{NULL}^t,(g_\emptyset,\emptyset))$. We show that $\mathfrak{G}[\![e]\!](\xi,\theta',G) = (p,G)$ by induction on e:

$e = x \in X$: $\mathfrak{G}[\![x]\!](\xi,\theta',G) = (\xi(x),G) = \mathfrak{G}[\![x]\!](\xi,\theta,G)$

$e = F \in DF$: $\mathfrak{G}[\![F]\!](\xi,\theta',G) = G + \langle F\rangle = \mathfrak{G}[\![F]\!](\xi,\theta,G)$

$e = f \in \Omega$: $\mathfrak{G}[\![f]\!](\xi,\theta',G) = G + \langle f\rangle = \mathfrak{G}[\![f]\!](\xi,\theta',G)$

$e = (e_0\ e_1\ \ldots\ e_m)$: Let
$$\begin{aligned}(p_0,(g_0,L_0)) &= \mathfrak{G}[\![e_0]\!](\xi,\theta,(g,L))\\ (p_j,(g_j,L_j)) &= \mathfrak{G}[\![e_j]\!](\xi,\theta,(g_{j-1},L_{j-1})) \text{ for } 1 \le j \le m\\ (p'_0,(g_0,L_0)) &= \mathfrak{G}[\![e_0]\!](\xi,\theta',(g,L))\\ (p'_j,(g_j,L_j)) &= \mathfrak{G}[\![e_j]\!](\xi,\theta',(g_{j-1},L_{j-1})) \text{ for } 1 \le j \le m\end{aligned}$$

We distinguish between partial and saturated applications:

1. If $1 \le m < n$, then we have

$$\mathfrak{G}[\![e]\!](\xi,\theta,(g,L)) = \mathsf{mkgarb}(L,(g_m,L_m) + \langle @,p_0,p_1,\ldots,p_m\rangle)$$

and conversely,

$$\mathfrak{G}[\![e]\!](\xi,\theta',(g,L)) = \mathsf{mkgarb}(L,(g_m,L_m) + \langle @,p'_0,p'_1,\ldots,p'_m\rangle)$$

We conclude with induction hypothesis and Corollaries 6.6 and 6.7.

2. If $m = n$, then we have

$$\begin{array}{rl} & \mathfrak{G}[\![e]\!](\xi, \theta, (g, L)) \\ = & \mathsf{mkgarb}(L, \mathsf{exec}^{t_1,\ldots,t_m \to t}(\theta, p_0, (g_m, L_m))(p_1, \ldots, p_m, (g_m, L_m))) \end{array}$$

and conversely,

$$\begin{array}{rl} & \mathfrak{G}[\![e]\!](\xi, \theta', (g, L)) \\ = & \mathsf{mkgarb}(L, \mathsf{exec}^{t_1,\ldots,t_m \to t}(\theta', p'_0, (g_m, L_m))(p'_1, \ldots, p'_m, (g_m, L_m))) \end{array}$$

We conclude with the induction hypothesis, Corollary 6.7, and Lemma 6.5.

$e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: Let $(p', (g', L')) = \mathfrak{G}[\![e_0]\!](\xi, \theta, G)$. We distinguish three cases:

1. If $p' \notin \mathsf{Dom}(g')$ then $\mathfrak{G}[\![\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](\xi, \theta, (g, L)) = (\mathsf{NULL}^t, (g_\emptyset, \emptyset))$ and this cannot happen.
2. If $p' \in \mathsf{Dom}(g')$ with $g'(p') = \langle \texttt{true} \rangle$ then we have

$$\mathfrak{G}[\![\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](\xi, \theta, (g, L)) = \mathsf{mkgarb}(L, \mathfrak{G}[\![e_1]\!](\xi, \theta, (g', L')))$$

   Hence, $\mathfrak{G}[\![e_1]\!](\xi, \theta, (g', L'))) \neq (\mathsf{NULL}^t, (g_\emptyset, \emptyset))$, and by induction hypothesis we know that $\mathfrak{G}[\![e_1]\!](\xi, \theta', (g', L')) = \mathfrak{G}[\![e_1]\!](\xi, \theta, (g', L'))$. We conclude with Corollary 6.7.
3. If $p' \in \mathsf{Dom}(g')$ with $g'(p') = \langle \texttt{false} \rangle$ then we can conclude analogous to the previous case. q.e.d.

We use this lemma to show that the sequence of successive applications of $\Phi_{\mathfrak{G},P}$ is componentwise convergent.

LEMMA 6.7
Let P be a program. The sequence $(\Phi^n_{\mathfrak{G},P}(T_b))_{n \in \mathbb{N}}$ is componentwise convergent. □

PROOF Let $P = (F(x_{j1}, \ldots, x_{jn_j}) := e_j \mid 1 \leq j \leq p)$ with $F_j \in DF^{t_{j1},\ldots,t_{jn_j} \to t_j}$, $x_{ji} \in X^{t_{ji}}$, and $e_j \in E^{t_j}$ with variables $\{x_{j1}, \ldots, x_{jn_j}\}$ for $1 \leq j \leq p$. We have to show that the sequence $((\Phi^n_{\mathfrak{G},P}(T_b))_j)_{n \in \mathbb{N}}$ is convergent for all $1 \leq j \leq p$. Therefore, we must find an index i for all $(g_1, \ldots, g_p) \in \mathsf{FS}_{\mathfrak{G},P}$ such that for all $1 \leq k < \infty$ holds:

$$((\Phi^i_{\mathfrak{G},P}(T_b))_j)(g_1, \ldots, g_p) = ((\Phi^{i+k}_{\mathfrak{G},P}(T_b))_j)(g_1, \ldots, g_p)$$

Let $(g_1, \ldots, g_p) \in \mathsf{FS}_{\mathfrak{G},P}$. We distinguish two cases:

1. If $((\Phi^i_{\mathfrak{G},P}(T_b))_j)(g_1, \ldots, g_p) = (T_b)_j$ for all $i \in \mathbb{N}$ then we choose the index 0.
2. If there exists an index $i \in \mathbb{N}$ such that $((\Phi^i_{\mathfrak{G},P}(T_b))_j))(g_1, \ldots, g_p) \neq (T_b)_j$, then this is equivalent to $\mathfrak{G}[\![e_j]\!](\xi, \theta, G) = (p_r, G_r) \neq (\mathsf{NULL}^{t_j}, (g_\emptyset, \emptyset))$ for all environments $\xi = [x_{j1}/p_1, \ldots, x_{jn_j}/p_n]$ and all graphs G where $\theta := [F_1/g_1, \ldots, F_p/g_p]$. The preceding lemma (Lemma 6.6) guarantees that

$$\mathfrak{G}[\![e_j]\!](\xi, \theta', G) = (p_r, G_r)$$

for all "larger" environments θ'. Hence, we have

$$((\Phi^{i+k}_{\mathfrak{G},P}(T_b))_j))(g_1,\ldots,g_p) = \lambda(p_1,\ldots,p_{n_j},G).(p_r,G_r)$$

for all $k \in \mathbb{N}$. q.e.d.

Note that the determination of the index is not constructive and hence the result does not yield a decision procedure for the test if a function is defined at certain points.

We conclude this section by the definition of the graph program semantics.

Definition 6.22 (Graph Program Semantics $\mathfrak{G}$)
Given a program $P = (F_j(x_{j1},\ldots,x_{jn_j}) := e_j \mid 1 \leq j \leq p)$ with $F_j \in DF^{t_{j1},\ldots,t_{jn_j}\to t_j}$, $x_{ji} \in X^{t_{ji}}$, and $e_j \in E^{t_j}$ with variables $\{x_{j1},\ldots,x_{jn_j}\}$ for $1 \leq j \leq p$ the *graph semantics of* P is an environment $\mathfrak{G}[\![P]\!] \in \mathsf{Env}(DF, \mathsf{GOps})$ given by

$$\mathfrak{G}[\![P]\!] := [F_1/\mathsf{fix}(\Phi_{\mathfrak{G},P})_1,\ldots,F_p/\mathsf{fix}(\Phi_{\mathfrak{G},P})_p]$$

Here, $\mathsf{fix}(\Phi_{\mathfrak{G},P})$ is the limit of the transformation $\Phi_{\mathfrak{G},P}$ defined as

$$\mathsf{fix}(\Phi_{\mathfrak{G},P}) = \lim_{n\to\infty} \Phi^n_{\mathfrak{G},P}(T_b) \qquad \triangleleft$$

Example: For the `append` program $P_{\texttt{append}}$, we have the following transformation

$$\Phi_{\mathfrak{G},P_{\texttt{append}}}(g) = \lambda(p,p',G).\mathfrak{G}[\![\texttt{if (is-Nil l) ...}]\!]([\texttt{l}/p,\texttt{l'}/p'],[\texttt{append}/g],G)$$

Successive application of $\Phi_{\mathfrak{G},P_{\texttt{append}}}$ on the least element

$$T_b = \lambda(p,p',G).(\mathsf{NULL}^{t_{\texttt{ListOfInt}}},(g_\emptyset,\emptyset))$$

of the associated function space $\mathsf{FS}_{\mathfrak{G},P_{\texttt{append}}} = \mathsf{GOps}^{\texttt{ListOfInt},\texttt{ListOfInt}\to\texttt{ListOfInt}}$ yields the following sequence:

$$\Phi^1_{\mathfrak{G},P_{\texttt{append}}}(T_b) = (p,p',(g,L)) \mapsto \begin{cases} (p',(g[\langle\texttt{is-Nil}\rangle,\langle\texttt{true}\rangle],L)) & \text{if } g(p) = \langle\texttt{Nil}\rangle \\ (\mathsf{NULL}^{t_{\texttt{ListOfInt}}},(g_\emptyset,\emptyset)) & \text{otherwise} \end{cases}$$

$$\Phi^2_{\mathfrak{G},P_{\texttt{append}}}(T_b) = (p,p',(g,L)) \mapsto$$
$$\begin{cases} (p',(g[\langle\texttt{is-Nil}\rangle,\langle\texttt{true}\rangle],L)) & \text{if } g(p) = \langle\texttt{Nil}\rangle \\ (g[\langle\texttt{is-Nil}\rangle,\langle\texttt{false}\rangle,\langle\texttt{is-Nil}\rangle,\langle\texttt{true}\rangle],L) + \langle\texttt{Cons},p_1,p'\rangle & \\ & \text{if } g(p) = \langle\texttt{Cons},p_1,q_1\rangle,\ g(q_1) = \langle\texttt{Nil}\rangle \\ (\mathsf{NULL}^{t_{\texttt{ListOfInt}}},(g_\emptyset,\emptyset)) & \text{otherwise} \end{cases}$$

$$\vdots$$

$$\Phi^m_{\mathfrak{G},P_{\texttt{append}}}(T_b) = (p,p',(g,L)) \mapsto$$
$$\begin{cases} (p',(g[\langle\texttt{is-Nil}\rangle,\langle\texttt{true}\rangle],L)) & \text{if } g(p) = \langle\texttt{Nil}\rangle \\ (g[\langle\texttt{is-Nil}\rangle,\langle\texttt{false}\rangle,\langle\texttt{is-Nil}\rangle,\langle\texttt{true}\rangle],L) & \text{if } g(p) = \langle\texttt{Cons},p_1,q_1\rangle, \\ +\langle\texttt{Cons},p_1,p'\rangle & \quad g(q_1) = \langle\texttt{Nil}\rangle \\ \vdots & \\ (r_1,(g\,[\langle\texttt{is-Nil}\rangle,\langle\texttt{false}\rangle,\ldots,\langle\texttt{is-Nil}\rangle,\langle\texttt{true}\rangle] & \text{if } \ g(p) = \langle\texttt{Cons},p_1,q_1\rangle, \\ \qquad [r_1/\langle\texttt{Cons},p_1,r_2\rangle,\ldots,r_{m-1}/\langle\texttt{Cons},p_{m-1},p'\rangle] & \quad g(p_i) = \langle\texttt{Cons},p_{i+1},q_{i+1}\rangle, \\ \quad L\cup\{r_1,\ldots,r_{m-1}\})) & \quad g(q_{m-1}) = \langle\texttt{Nil}\rangle \\ (\mathsf{NULL}^{t_{\texttt{ListOfInt}}},(g_\emptyset,\emptyset)) & \text{otherwise} \end{cases}$$

Here, the notation $g[hn_1, \ldots, hn_2]$ denotes a graph function where the additional heap nodes hn_i are allocated corresponding to the allocation strategy. All these node are garbage nodes.

In the m–th iteration, lists of length $n < m$ as first argument are copied, producing n new constructor cells for `Cons`. Furthermore, $2n$ garbage cells for the function `is−Nil`, $2(n-1)$ garbage cells for the constant `false`, and one garbage cell for the constant `true` are produced. The garbage cells are created by the condition `(is−Nil l)`. $\diamond$

6.6 Soundness of $\mathfrak{G}$ wrt. $\mathfrak{M}$

Our next aim is to prove that our model of graph reduction is sound wrt. the denotational semantics. We establish this soundness by using the representation function rep from Definition 6.5. Of course, we cannot expect $\mathfrak{G}$ to be complete wrt. $\mathfrak{M}$, because while $\mathfrak{M}$ is a non–strict semantics, $\mathfrak{G}$ is strict. The main result of this section is that if $\mathfrak{G}$ produces a result then it represents the result from $\mathfrak{M}$.

To cater for all intermediate steps, we introduce soundness notions for values, variable environments, (tuples of) graph operations, and environments for defined functions.

Definition 6.23 (Soundness)

1. Let $(g, L) \in \mathsf{G}$ be a graph, $l \in \mathsf{Dom}(g^t)$ be a location, $\sigma \in \mathsf{Env}(DF, \mathsf{Ops})$ be an environment for defined functions, and $v \in \mathsf{CT}^t$ be a value. The tuple $(l, (g, L))$ is *sound wrt. v under σ* iff $l \in \mathsf{Dom}(g)$ implies that $\mathsf{rep}_g(l, \sigma) = v$.

2. Let $(g, L) \in \mathsf{G}$ be a graph and $\sigma \in \mathsf{Env}(DF, \mathsf{Ops})$ be an environment for defined functions. An environment for variables $\xi \in \mathsf{Env}(X, \mathsf{P})$ is *sound wrt. an environment $\beta \in \mathsf{Env}(X, \mathsf{CT})$ under σ* iff for all $x \in X$ holds that $(\xi(x), G)$ is sound wrt. $\beta(x)$ under σ.

3. Let $\sigma \in \mathsf{Env}(DF, \mathsf{Ops})$ be an environment for defined functions. A function $f^G : \mathsf{P}^{t_1} \times \cdots \times \mathsf{P}^{t_n} \times \mathsf{G} \to \mathsf{P}^t \times \mathsf{G}$ is *sound wrt. $f : \mathsf{CT}^{t_1}, \ldots, \mathsf{CT}^{t_n} \to \mathsf{CT}^t$ under σ* iff for all graphs $(g, L) \in \mathsf{G}$, locations $l_i \in \mathsf{Dom}(g^{t_i})$, and values $v_i \in \mathsf{CT}^{t_i}$ with $(l_i, (g, L))$ sound wrt. v_i under σ $(1 \leq i \leq n)$ holds: $f^G(l_1, \ldots, l_n, (g, L))$ is sound wrt. $f(v_1, \ldots, v_n)$ under σ.

4. Let $\sigma \in \mathsf{Env}(DF, \mathsf{Ops})$ be an environment for defined functions. A tuple of functions $(f_1^G, \ldots, f_n^G)$, $f_i^G : \mathsf{P}^{t_{i,1}} \times \cdots \times \mathsf{P}^{t_{i,n_i}} \times \mathsf{G} \to \mathsf{P}^{t_i} \times \mathsf{G}$ is *sound wrt. $(f_1, \ldots, f_n)$, $f_i : \mathsf{CT}^{t_{i,1}}, \ldots, \mathsf{CT}^{t_{i,n_i}} \to \mathsf{CT}^{t_i}$ under σ* iff f_i^G is sound wrt. f_i under σ for all $1 \leq i \leq n$.

5. An environment for defined functions $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ is *sound wrt. an environment for defined functions $\sigma \in \mathsf{Env}(DF, \mathsf{Ops})$* iff for all functions $F \in DF$ holds $\theta(F)$ is sound wrt. $\sigma(F)$ under σ. $\lhd$

The essential point in the definition is soundness of values. Here we use the representation function to relate a tuple of pointer and graph on one hand and a denotational value on the other hand. By definition all tuples which contain an invalid pointer (especially the undefined pointer NULL) are sound wrt. all values.

A problem arises since the graphs contain nodes with the *names* of functions. Therefore, we need an environment for defined functions which is used by rep to assign values to partial applications.

Our first result is that the graph semantics of intrinsic functions implements the denotation of intrinsic functions. Note that this result is stronger than soundness, because it does not allow the pointer to be invalid.

LEMMA 6.8
Let $f \in \Omega^{t_1,\ldots,t_n \to t}$ be an intrinsic function, $\sigma \in \mathsf{Env}(DF, \mathsf{Ops})$ an environment, $G \in \mathsf{G}$ a graph, and $l_i \in \mathsf{HL}^{t_i}$ locations with $v_i = \mathsf{rep}_g(l_i, \sigma)$ for $1 \le i \le n$. If $(l_r, (g_r, L_r)) = \mathfrak{G}[\![f]\!](l_1, \ldots, l_n, G)$ then we also have $\mathsf{rep}_g(l_r, \sigma) = \mathfrak{M}[\![f]\!](v_1, \ldots, v_n)$. □

PROOF trivial q.e.d.

We show that the auxiliary function exec implements application.

LEMMA 6.9 (exec IMPLEMENTS APPLICATION)
Let $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$, $\sigma \in \mathsf{Env}(DF, \mathsf{Ops})$ be environments for defined function such that θ is sound wrt. σ. Furthermore, let $l_f \in \mathsf{HL}^{t_1,\ldots,t_n \to t}$, $l_1 \in \mathsf{HL}^{t_1}, \ldots, l_n \in \mathsf{HL}^{t_n}$ be locations and $(g, L) \in \mathsf{G}$ be a graph such that $\mathsf{rep}_g(l_f, \sigma) = v_f$ and $\mathsf{rep}_g(l_i, \sigma) = v_i$ for $1 \le i \le n$. If $(l_r, (g_r, L_r)) = \mathsf{exec}^{t_1,\ldots,t_n \to t}(\theta, l_f, (g, L))(l_1, \ldots, l_n, (g, L))$ then $\mathsf{rep}_{g_r}(l_r, \sigma) = v_f(v_1, \ldots, v_n)$. □

PROOF Corresponding to the definition of exec, we distinguish four cases:

1. $l_f \notin \mathsf{Dom}(g)$. This cannot happen, because otherwise f would not be defined.
2. $g(l_f) = \langle f \rangle$, $f \in \Omega^{t_1,\ldots,t_n \to t}$: We have

$$\begin{aligned} (l_r, (g_r, L_r)) &= \mathsf{exec}^{t_1,\ldots,t_n \to t}(\theta, l_f, (g, L))(l_1, \ldots, l_n, (g, L)) \\ &= \mathfrak{G}[\![f]\!](l_1, \ldots, l_n, (g, L)) \end{aligned}$$

   On the other hand, we have $v_f = \mathsf{rep}_g(l_f, \sigma) = \mathfrak{M}[\![f]\!]([], \sigma) = \mathfrak{M}[\![f]\!]$ and we can conclude with Lemma 6.8.
3. $g(l_f) = \langle F \rangle$, $F \in DF^{t_1,\ldots,t_n \to t}$

$$\begin{aligned} (l_r, (g_r, L_r)) &= \mathsf{exec}^{t_1,\ldots,t_n \to t}(\theta, l_f, (g, L))(l_1, \ldots, l_n, (g, L)) \\ &= (\theta(F))(l_1, \ldots, l_n, (g, L)) \end{aligned}$$

   On the other hand, given that $v_f = \mathsf{rep}_g(l_f, \sigma) = \mathfrak{M}[\![F]\!]([], \sigma) = \sigma(F)$, we can conclude with θ is sound wrt. σ.
4. $g(l_f) = \langle @, p'_f, p'_1, \ldots, p'_m \rangle$, $p'_f \in \mathsf{P}^{t'_1,\ldots,t'_m,t_1,\ldots,t_n \to t}$, $p'_{m+i} = p_i$, $t'_{m_i} = t_i$ $(1 \le i \le n)$

$$\begin{aligned} (l_r, (g_r, L_r)) &= \mathsf{exec}^{t_1,\ldots,t_n \to t}(\theta, l_f, (g, L))(l_1, \ldots, l_n, (g, L)) \\ &= \mathsf{exec}^{t'_1,\ldots,t'_{m+n} \to t}(\theta, p'_f, (g, L))(p'_1, \ldots, p'_{m+n}, (g, L)) \end{aligned}$$

Furthermore, we have $v_f = \mathsf{rep}_g(l_f, \sigma) = \lambda(v_1, \ldots, v_n).v'_f(v'_1, \ldots, v'_m, v_1, \ldots, v_n)$ where $v'_f = \mathsf{rep}_g(p'_f, \sigma)$, and $v'_i = \mathsf{rep}_g(p'_i, \sigma)$ for $1 \leq i \leq m$. Because the graph is finite and not cyclic this reduction can only occur finitely many times until one of the other cases occurs. q.e.d.

We prove our soundness result for expressions, which is the major task in this section.

Theorem 6.3 (Soundness Of Expression Semantics)
Let $e \in E^t$ be an expression, $G \in \mathsf{G}$ be a graph, and let $\xi \in \mathsf{Env}(X, \mathsf{P})$, $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$, $\beta \in \mathsf{Env}(X, \mathsf{CT})$, and $\sigma \in \mathsf{Env}(DF, \mathsf{Ops})$ be environments such that the variable environment ξ is sound wrt. β under σ and the environment for defined function θ is sound wrt. σ:

$$\mathfrak{G}[\![e]\!](\xi, \theta, G) \text{ is sound wrt. } \mathfrak{M}[\![e]\!](\beta, \sigma) \text{ under } \sigma \qquad \square$$

Proof Let $G = (g, l)$.

Induction on e

$e = x \in X$: We have $\mathfrak{G}[\![x]\!](\xi, \theta, G) = (\xi(x), G)$ and hence soundness by the assumption that ξ is sound wrt. β.

$e = F \in DF$: We have $\mathfrak{G}[\![F]\!](\xi, \theta, G) = G + \langle F \rangle = (l', G')$ where $l' = \mathsf{free}^t(\mathsf{Dom}(g))$, and $G' = (g[l'/\langle F \rangle], L \cup \{l'\})$. Obviously, $l' \in \mathsf{Dom}(g[l'/\langle F \rangle])$ and by definition we have

$$\mathsf{rep}_{g[l'/\langle F \rangle]}(l, \sigma) = \mathfrak{M}[\![F]\!]([], \sigma) = \sigma(F) = \mathfrak{M}[\![F]\!](\beta, \sigma)$$

$e = f \in \Omega$: Similar to the previous case.

$e = (e_0\ e_1\ \ldots\ e_m)$: Let
$$\begin{aligned}(p_0, (g_0, L_0)) &= \mathfrak{G}[\![e_0]\!](\xi, \theta, (g, L)) \\ (p_j, (g_j, L_j)) &= \mathfrak{G}[\![e_j]\!](\xi, \theta, (g_{j-1}, L_{j-1})) \quad \text{for } 1 \leq j \leq m\end{aligned}$$

We distinguish two cases:

1. if $m = n$, i.e. the application is saturated:

$$\begin{aligned}&\mathfrak{G}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\xi, \theta, (g, L)) \\ &\quad = \mathsf{mkgarb}(L, \mathsf{exec}^{t_1, \ldots, t_m \to t}(\theta, p_0, (g_m, L_m))(p_1, \ldots, p_m, (g_m, L_m))) \\ &\quad = \begin{cases}(\mathsf{NULL}^t, (g_\emptyset, \emptyset)) & \text{if } l_r = \mathsf{NULL}^t \\ (l_r, (g_r, L \cup \overrightarrow{l_r})) & \text{otherwise}\end{cases} \\ &\text{with } (l_r, (g_r, L_r)) = \mathsf{exec}^{t_1, \ldots, t_m \to t}(\theta, p_0, (g_m, L_m))(p_1, \ldots, p_m, (g_m, L_m))\end{aligned}$$

If $l_r = \mathsf{NULL}^t$ then there is nothing more to prove. Otherwise, we have

$$\mathfrak{M}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\beta, \sigma) = (\mathfrak{M}[\![e_0]\!](\beta, \sigma))(\mathfrak{M}[\![e_1]\!](\beta, \sigma), \ldots, \mathfrak{M}[\![e_m]\!](\beta, \sigma))$$

Note that $l_i \neq \mathsf{NULL}^{t_i}$ for all i, because otherwise the strictness of $\mathfrak{G}$ would imply $l_r = \mathsf{NULL}^t$. We conclude with the induction hypothesis, Lemma 6.9, and the property that application of a sound function to sound arguments yields a sound result.

2. $1 \le m < n$, i.e. the application is partial:

$$\begin{aligned}\mathfrak{G}[\![(e_0\ e_1\ \dots\ e_m)]\!](\xi,\theta,(g,L)) &= \mathsf{mkgarb}(L,(g_m,L_m)+\langle @,p_0,p_1,\dots,p_m\rangle)\\ &= \begin{cases}(\mathsf{NULL}^t,(g_\emptyset,\emptyset)) & \text{if } l_r=\mathsf{NULL}^t\\ (l_r,(g_r,L\cup\overrightarrow{l_r})) & \text{otherwise}\end{cases}\\ \text{with } l_r &= \mathsf{free}^t(\mathsf{Dom}(g_m))\\ g_r &= g_m[l_r/\langle @,p_0,p_1,\dots,p_m\rangle]\\ L_r &= L_m\cup\{l_r\}\end{aligned}$$

If $l_r = \mathsf{NULL}^t$ then there is nothing more to prove. Otherwise, we have

$$\mathsf{rep}_{g_r}(l_r) = \lambda(v_{m+1},\dots,v_n).v_0(v_1,\dots,v_m,v_{m+1},\dots,v_n)$$

with $v_i = \mathsf{rep}_g(l_i,\sigma)$ for $0 \le i \le m$. Note that $l_i \ne \mathsf{NULL}^{t_i}$ for all i, because otherwise the strictness of $\mathfrak{G}$ would imply $l_r = \mathsf{NULL}^t$. We can conclude with the observation that

$$\begin{aligned}\mathfrak{M}[\![(e_0\ e_1\ \dots\ e_m)]\!](\beta,\sigma) \quad &= \quad \lambda(v_{m+1},\dots,v_n).v_0(v_1,\dots,v_m,v_{m+1},\dots,v_n)\\ &\qquad \text{where } v_i=\mathfrak{M}[\![e_i]\!](\beta,\sigma)\ 0\le i\le m\end{aligned}$$

$e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: Let $(p',(g',L')) := \mathfrak{G}[\![e_0]\!](\xi,\theta,G)$. We distinguish three cases:

1. $p' \notin \mathsf{Dom}(g')$. We have $\mathfrak{G}[\![\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](\xi,\theta,(g,L)) = (\mathsf{NULL}^t,(g_\emptyset,\emptyset))$ and by definition this is sound wrt. every value.
2. $p' \in \mathsf{Dom}(g')$, $g'(p') = \langle\texttt{true}\rangle$. Then we know by induction hypothesis that $\mathfrak{M}[\![e_0]\!](\beta,\sigma) = \texttt{true}$ and hence

$$\begin{aligned}\mathfrak{G}[\![\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](\xi,\theta,(g,L)) \quad &= \quad \mathsf{mkgarb}(L,\mathfrak{G}[\![e_1]\!](\xi,\theta,(g',L')))\\ &= \quad \begin{cases}(\mathsf{NULL}^t,(g_\emptyset,\emptyset)) & \text{if } l_r=\mathsf{NULL}^t\\ (l_r,(g_r,L\cup\overrightarrow{l_r})) & \text{otherwise}\end{cases}\\ \text{with } (l_r,(g_r,L_r)) \quad &= \quad \mathfrak{G}[\![e_1]\!](\xi,\theta,(g',L'))\end{aligned}$$

   If $l_r = \mathsf{NULL}^t$ then there is nothing more to prove. Otherwise, we have

$$\mathfrak{M}[\![\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](\beta,\sigma) = \mathfrak{M}[\![e_1]\!](\beta,\sigma)$$

   We can conclude with the induction hypothesis for the expression e_1 and the graph $(g_r, L \cup \overrightarrow{l_r})$.
3. $p' \in \mathsf{Dom}(g')$, $g'(p') = \langle\texttt{false}\rangle$. Similar to the previous case. q.e.d.

Our main result is the next theorem, which establishes soundness of the graph semantics.

THEOREM 6.4 (SOUNDNESS OF $\mathfrak{G}$)
For all programs P holds that $\mathfrak{G}[\![P]\!]$ is sound wrt. $\mathfrak{M}[\![P]\!]$. □

PROOF By definition, we know that

$$\begin{aligned}
\mathfrak{G}[\![P]\!] &= [F_1/\mathsf{fix}_{\mathfrak{G}}(\Phi_{\mathfrak{G},P})_1, \ldots, F_p/\mathsf{fix}_{\mathfrak{G}}(\Phi_{\mathfrak{G},P})_p] \\
&\quad \text{with } \mathsf{fix}_{\mathfrak{G}}(\Phi_{\mathfrak{G},P}) = \lim_{n\to\infty} \Phi^n_{\mathfrak{G},P}(T_b) \\
\mathfrak{M}[\![P]\!] &= [F_1/\mathsf{fix}_{\mathfrak{M}}(\Phi_{\mathfrak{M},P})_1, \ldots, F_p/\mathsf{fix}_{\mathfrak{M}}(\Phi_{\mathfrak{M},P})_p] \\
&\quad \text{with } \mathsf{fix}_{\mathfrak{M}}(\Phi_{\mathfrak{M},P}) = \bigsqcup\{\Phi^i_{\mathfrak{M},P}(\perp^{\mathsf{FS}_{\mathfrak{M},P}}) \mid i \in \mathbb{N}\}
\end{aligned}$$

By Lemma 6.3 we know that the following relation holds:

$$\mathsf{fix}_{\mathfrak{G}}(\Phi_{\mathfrak{G},P}) = \lim_{n\to\infty} \Phi^n_{\mathfrak{G},P}(T_b) \in \mathsf{lub}(\{\Phi^i_{\mathfrak{G},P}(T_b) \mid i \in \mathbb{N}\})$$

Hence it suffices to show that for all $T \in \mathsf{lub}(\{\Phi^i_{\mathfrak{G},P}(T_b) \mid i \in \mathbb{N}\})$ holds that T is sound wrt. $\mathfrak{M}[\![P]\!]$. Since this relation in continuous, it suffices to validate the more general relation that $\Phi^i_{\mathfrak{G},P}(T_b)$ is sound wrt. $\Phi^i_{\mathfrak{M},P}(\perp^{\mathsf{FS}_{\mathfrak{M},P}})$ for all $i \in \mathbb{N}$. This immediately follows by induction from Theorem 6.3. q.e.d.

6.7 Escaping as Graph Property

In this section we investigate the relation between the augmented semantics presented in Chapter 3 and the graph semantics. We show that the augmented semantics is indeed a model of the escape behaviour of the graph semantics: A value escapes from an expression e corresponding to the augmented semantics iff the corresponding graph nodes are reachable from the result of evaluating e using the graph semantics. By showing this, we demonstrate that we have chosen a reasonable approach, which allows the prediction of the behaviour of implementations based on denotational semantics.

We formalise this intuition by the notion of escape–soundness. Again, we have to cater for all intermediate steps; therefore, we defined escape–soundness not only for environments of defined functions, but also for values, graph operations, and variable environments.

DEFINITION 6.24 (ESCAPE–SOUNDNESS)
We simultaneously define the following notions:

1. Let $(g, L) \in \mathsf{G}$ be a graph, $l \in \mathsf{Dom}(g^t)$ be a location, $\widehat{v} \in \widehat{\mathsf{CT}}^t$ be a value, and $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ be an environment. We define that the tuple $(l, (g, L))$ is *escape–sound wrt. v under θ* by induction on the type: either $t \in S$ or $t = t_1, \ldots, t_n \to t_0$ and the function $\mathsf{exec}^{t_1,\ldots,t_n\to t_0}(\theta, l, (g, L))$ is escape–sound wrt. $\widehat{v}$ under θ (see Item 2).

2. Let $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ be an environment. A function $f : \mathsf{GOps}^{t_1,\ldots,t_n\to t}$ is *escape–sound wrt. $\widehat{f} \in \widehat{\mathsf{Ops}}^{t_1,\ldots,t_n\to t}$ under θ* iff for all graphs $(g, L) \in \mathsf{G}$, locations $l_i \in \mathsf{Dom}(g^{t_i})$, and values $\widehat{v_i} \in \widehat{\mathsf{CT}}^{t_i}$ with $(l_i, (g, L))$ escape–sound wrt. $\widehat{v_i}$ under θ for $1 \le i \le n$ holds: if we have $(l_r, (g_r, L_r)) = f(l_1, \ldots, l_n, (g, L))$ then

   (a) $(l_r, (g_r, L_r))$ is escape–sound wrt. $\widehat{f}(\widehat{v}_1, \ldots, \widehat{v}_n)$ under θ.

(b) $l_r \neq \mathsf{NULL}^t$ implies that parts of $\widehat{v}_j$ escape from the application $\widehat{f}(\widehat{v}_1, \ldots, \widehat{v}_n)$ iff $\overleftrightarrow{l_i} \cap \overleftrightarrow{l_r} \neq \emptyset$.

3. Let $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ be an environment and let $(g, L) \in \mathsf{G}$ be a graph. An environment for variables $\xi \in \mathsf{Env}(X, \mathsf{P})$ is *escape–sound wrt.* $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ *under* θ *and* G iff for all $x \in X$ holds that $(\xi(x), G)$ is escape–sound wrt. $\widehat{\beta}(x)$ under θ.

4. An environment for defined functions $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ is *escape–sound wrt.* an environment $\widehat{\sigma} \in \mathsf{Env}(DF, \widehat{\mathsf{Ops}})$ iff for all functions $F \in DF$ holds that $\theta(F)$ is escape–sound wrt. $\widehat{\sigma}(F)$ under θ. $\lhd$

Remarks:

- Although Items 1 and 2 are mutually dependent, the notions are well–defined: Each step reduces the size of the types.
- The central point is the definition of escape–sound functions. It essentially means that escaping as defined in Chapter 3 is captured by the graph function.
- Here, we solely consider escaping, because the aspect that the computed values are sound was already handled in the previous section.

We prove our soundness result for expressions, which is the major task in this section.

Theorem 6.5 (Escape–Soundness Of Expression Semantics)
Let $e \in E^t$ be an expression, $x \in X$ be a variable, $G \in \mathsf{G}$ be a graph, $\xi \in \mathsf{Env}(X, \mathsf{HL})$, $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ be graph environments, and $\widehat{\beta} \in \mathsf{Env}(X, \widehat{\mathsf{CT}})$, $\widehat{\sigma} \in \mathsf{Env}(DF, \widehat{\mathsf{Ops}})$ be augmented environments such that ξ is escape–sound wrt. $\widehat{\beta}$ under θ and G, and θ is escape–sound wrt. $\widehat{\sigma}$. Then we have that the value $(l_r, (g_r, L_r)) := \mathfrak{G}[\![e]\!](\xi, \theta, G)$ is escape–sound wrt. $\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma})$ under θ and (g_r, L_r). Furthermore, if $l_r \neq \mathsf{NULL}^t$ then parts of x escape from e under $\widehat{\sigma}$ iff $\overleftrightarrow{\xi(x)} \cap \overleftrightarrow{l_r} \neq \emptyset$. $\square$

Proof Assume that $(l_r, (g_r, L_r)) = \mathfrak{G}[\![e]\!](\xi, \theta, G)$ with $l_r \neq \mathsf{NULL}^t$ and let $x \in X$ be such that $\widehat{\beta}(y)$ has void augmentation iff $x \neq y$. By Definition 3.11, parts of x escape from e under $\widehat{\sigma}$ iff $\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma})$ has non–void augmentation. We show the second by induction on e:

$e = x \in X$: We have $\widehat{\mathfrak{M}}[\![x]\!](\widehat{\beta}, \widehat{\sigma}) = \widehat{\beta}(x)$ and hence $\widehat{\mathfrak{M}}[\![x]\!](\widehat{\beta}, \widehat{\sigma})$ always has non–void augmentation. On the other hand, $\mathfrak{G}[\![e]\!](\xi, \theta, G) = (\xi(x), G)$ and hence $l_r = \xi(x)$, which trivially implies $\overleftrightarrow{\xi(x)} \cap \overleftrightarrow{l_r} = \overleftrightarrow{l_r} \ni l_r$. Furthermore, $\xi(x)$ is escape–sound wrt. $\widehat{\beta}(x)$ under θ and (g_r, L_r) by assumption.

$e = F \in DF$: By definition, $\widehat{\mathfrak{M}}[\![F]\!] = \widehat{\beta}(F)$ and hence no part of x escapes. Similarly, $\mathfrak{G}[\![F]\!](\xi, \theta, G) = G + \langle F \rangle$ and therefore we always have $\overleftrightarrow{\xi(x)} \cap \overleftrightarrow{l_r} = \emptyset$.

Given that, it remains to show that $G + \langle F \rangle$ is escape–sound wrt. $\widehat{\beta}(F)$ under θ and (g_r, L_r). Let $(b, \widehat{f}) := \widehat{\beta}(F)$. By Definition 6.24 (Item 2), we have to show that the function $f := \mathsf{exec}^{t_1, \ldots, t_n \to t}(\theta, l_r, (g_r, L_r)) = \theta(F)$ is escape–sound wrt. $\widehat{f}$ under θ. However, this is true by assumption.

$e = f \in \Omega$: Like the previous case.

$e = (e_0\ e_1\ \ldots\ e_m)$: Let
$$\begin{array}{rcl} (b, \widehat{f}) & = & \widehat{\mathfrak{M}}[\![e_0]\!](\widehat{\beta}, \widehat{\sigma}) \\ (p_0, (g_0, L_0)) & = & \mathfrak{G}[\![e_0]\!](\xi, \theta, (g, L)) \\ (p_j, (g_j, L_j)) & = & \mathfrak{G}[\![e_j]\!](\xi, \theta, (g_{j-1}, L_{j-1})) \quad \text{for } 1 \leq j \leq m \end{array}$$

We distinguish two cases: either the application is saturated with $m = n$ or it is partial with $m < n$.

$m = n$: $\mathfrak{G}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\xi, \theta, (g, L))$
$$\begin{array}{l} = \mathsf{mkgarb}(L, \mathsf{exec}^{t_1,\ldots,t_m \to t}(\theta, p_0, (g_m, L_m))(p_1, \ldots, p_m, (g_m, L_m))) \\ = \begin{cases} (\mathsf{NULL}^t, (g_\emptyset, \emptyset)) & \text{if } l_r = \mathsf{NULL}^t \\ (l_r, (g_r, L \cup \overrightarrow{l_r})) & \text{otherwise} \end{cases} \end{array}$$
with $(l_r, (g_r, L_r)) = \mathsf{exec}^{t_1,\ldots,t_m \to t}(\theta, p_0, (g_m, L_m))(p_1, \ldots, p_m, (g_m, L_m))$

If $l_r = \mathsf{NULL}^t$ then there is nothing more to prove. Otherwise,

$$\widehat{\mathfrak{M}}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\widehat{\beta}, \widehat{\sigma}) = \widehat{f}(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \ldots, \widehat{\mathfrak{M}}[\![e_n]\!](\widehat{\beta}, \widehat{\sigma}))$$

Note that $l_i \neq \mathsf{NULL}^{t_i}$ for all i, because otherwise the strictness of $\mathfrak{G}$ would imply $l_r = \mathsf{NULL}^t$. We conclude this case with the induction hypothesis and that the application of escape–sound function to escape–sound arguments yields escape–sound results.

$m < n$: For the annotated semantics, we have $\widehat{\mathfrak{M}}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\widehat{\beta}, \widehat{\sigma}) = (b, \widehat{f}')$ with $\widehat{f}' = \lambda(\widehat{v}_{m+1}, \ldots, \widehat{v}_n).\widehat{f}(\widehat{\mathfrak{M}}[\![e_1]\!](\widehat{\beta}, \widehat{\sigma}), \ldots, \widehat{\mathfrak{M}}[\![e_m]\!](\widehat{\beta}, \widehat{\sigma}), \widehat{v}_{m+1}, \ldots, \widehat{v}_n)$. Hence we can conclude that

$$\begin{array}{ll} & (b, f') \text{ has non–void augmentation} \\ \iff & b = 1 \text{ or there exist } \widehat{v}_{m+1}, \ldots, \widehat{v}_n \text{ such that } \widehat{v}_i \in \widehat{\mathsf{CT}}^{t_i} \text{ has void augmentation} \\ & \text{for } 1 \leq i \leq m \text{ and } \widehat{f}'(\widehat{v}_{m+1}, \ldots, \widehat{v}_n) \text{ has non–void augmentation} \\ \iff & b = 1 \text{ or there exist } 1 \leq j \leq m \text{ such that } \widehat{\mathfrak{M}}[\![e_j]\!](\widehat{\beta}, \widehat{\sigma}) \text{ has non–void} \\ & \text{augmentation} \\ \iff & \text{there exist } \widehat{\mathfrak{M}}[\![e_j]\!](\widehat{\beta}, \widehat{\sigma}) \text{ with non–void augmentation} \end{array}$$

On the other hand, for the graph reduction semantics we have

$$\begin{array}{rl} \mathfrak{G}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\xi, \theta, (g, L)) & = \mathsf{mkgarb}(L, (g_m, L_m) + \langle @, p_0, p_1, \ldots, p_m \rangle) \\ & = \begin{cases} (\mathsf{NULL}^t, (g_\emptyset, \emptyset)) & \text{if } l_r = \mathsf{NULL}^t \\ (l_r, (g_r, L \cup \overrightarrow{l_r})) & \text{otherwise} \end{cases} \\ \text{with } l_r & = \mathsf{free}^t(\mathsf{Dom}(g_m)) \\ g_r & = g_m[l_r / \langle @, p_0, p_1, \ldots, p_m \rangle] \\ L_r & = L_m \cup \{l_r\} \end{array}$$

If $l_r = \mathsf{NULL}^t$ then there is nothing more to prove. Otherwise,

$$\overrightarrow{l_r} = \{l_r\} \cup \overrightarrow{p_0} \cup \cdots \cup \overrightarrow{p_m}$$

By induction hypothesis, we have that $\widehat{\mathfrak{M}}[\![e_j]\!](\widehat{\beta}, \widehat{\sigma})$ has non–void augmentation iff there exists $1 \leq j \leq m$ such that $\overrightarrow{\xi(x)} \cap \overrightarrow{p_j} \neq \emptyset$. This is equivalent to $\overrightarrow{\xi(x)} \cap \overrightarrow{l_r} \neq \emptyset$.

It remains to be shown that $(l_r, (g_r, L_r)$ is escape–sound wrt. $(b, \widehat{f'})$ under θ and (g_r, L_r). By Definition 6.24 (Item 2), we have to show that the function f'

$$\begin{aligned} f' &= \mathsf{exec}^{t_1,\ldots,t_m\to t}(\theta, l_r, (g_r, L_r)) \\ &= \lambda(p_{m+1},\ldots,p_n,G).f(p_1,\ldots,p_m,p_{m+1},\ldots,p_n,G) \\ &\ \text{where } f = \mathsf{exec}^{t_1,\ldots,t_n\to t}(\theta, p_0, (g, L)) \end{aligned}$$

is escape–sound wrt. $\widehat{f'}$ under θ. This is true by induction hypothesis.

$e =$ **if** e_0 **then** e_1 **else** e_2: Trivial. q.e.d.

This theorem implies the corresponding result for programs.

THEOREM 6.6 (ESCAPE–SOUNDNESS OF $\mathfrak{G}$)
For all programs P holds that $\mathfrak{G}[\![P]\!]$ is escape–sound wrt. $\widehat{\mathfrak{M}}[\![P]\!]$. □

PROOF By definition, we know that

$$\begin{aligned} \mathfrak{G}[\![P]\!] &= [F_1/\mathsf{fix}_{\mathfrak{G}}(\Phi_{\mathfrak{G},P})_1,\ldots,F_p/\mathsf{fix}_{\mathfrak{G}}(\Phi_{\mathfrak{G},P})_p] \\ &\ \text{with } \mathsf{fix}_{\mathfrak{G}}(\Phi_{\mathfrak{G},P}) = \lim_{n\to\infty}\Phi^n_{\mathfrak{G},P}(T_b) \\ \widehat{\mathfrak{M}}[\![P]\!] &= [F_1/\mathsf{fix}_{\widehat{\mathfrak{M}}}(\Phi_{\widehat{\mathfrak{M}},P})_1,\ldots,F_p/\mathsf{fix}_{\widehat{\mathfrak{M}}}(\Phi_{\widehat{\mathfrak{M}},P})_p] \\ &\ \text{with } \mathsf{fix}_{\widehat{\mathfrak{M}}}(\Phi_{\widehat{\mathfrak{M}},P}) = \bigsqcup\{\Phi^i_{\widehat{\mathfrak{M}},P}(\bot^{\mathsf{FS}_{\widehat{\mathfrak{M}},P}}) \mid i\in\mathbb{N}\} \end{aligned}$$

By Lemma 6.3 we know that the following relation holds:

$$\mathsf{fix}_{\mathfrak{G}}(\Phi_{\mathfrak{G},P}) = \lim_{n\to\infty}\Phi^n_{\mathfrak{G},P}(T_b) \in \mathsf{lub}(\{\Phi^i_{\mathfrak{G},P}(T_b) \mid i\in\mathbb{N}\})$$

Hence it suffices to show that for all $T \in \mathsf{lub}(\{\Phi^i_{\mathfrak{G},P}(T_b) \mid i\in\mathbb{N}\})$ holds that T is escape–sound wrt. $\widehat{\mathfrak{M}}[\![P]\!]$. Because this relation in continuous, we can prove this by validating the more general relation that $\Phi^i_{\mathfrak{G},P}(T_b)$ is escape–sound wrt. $\Phi^i_{\widehat{\mathfrak{M}},P}(\bot^{\mathsf{FS}_{\widehat{\mathfrak{M}},P}})$ for all $i\in\mathbb{N}$. This immediately follows by induction from Theorem 6.5. q.e.d.

With the last result we have obtained a characterisation of escaping in terms of graph reduction. Caused by sharing, however, non–escaping of parts does not imply that these parts are always garbage. Consider the program

```
  F x = (H (G x) x)
  G y = 2
H x y = x
```

Obviously, `x` does not escape from `G` but it is not garbage at the end of the evaluation of `G`. On the other hand, the evaluation of `G` creates a heap node for the value `2` which does not escape from `H`. Such newly created heap nodes cannot be shared and are therefore garbage at the end of the evaluation of `H`.

6.8 Summary

In this chapter, we have defined a denotational model of graph reduction. The main idea is to model the denotational view of functions on terms by functions on locations and graphs: $\mathsf{P}^{t_1} \times \cdots \times \mathsf{P}^{t_n} \times \mathsf{G} \to \mathsf{P}^t \times \mathsf{G}$. The parameter of type G is the graph containing the arguments to the function. The arguments are represented by pointers into the graph. The result is a tuple consisting of the modified graph and the pointer to the result in that graph. Since the graph is a unique component, we can easily implement this denotational graph semantics by an abstract machine where the graph is a global mutable component.

To model garbage in the graphs, we had to define the graph domain G as a quasi ordered set: The order on the graphs $\preccurlyeq$ had to ignore garbage, because otherwise removal of garbage would not be a monotonic operation. Because the standard theory of denotational semantics requires complete partially ordered sets as denotational domains, we have extended this theory accordingly. The generalised fixpoint theorem exploits the special structure of function spaces and uses a new notion of convergence. Like the standard theorem, it guarantees that we obtain a least fixpoint by successive application to a least element.

This approach came to fruition for the proofs of two major results of this chapter. We have shown that (1) the graph semantics $\mathfrak{G}$ is sound with respect to the reference semantics $\mathfrak{M}$ and (2) escaping in the augmented semantics $\widehat{\mathfrak{M}}$ is a precise model of reachability from the result in the graph semantics. Since our approach has avoided the translation of recursion into iteration both result were obtained essentially by induction on the structure of expressions.

7. Applications

In this chapter, we introduce two applications based on the knowledge of non–escaping of expressions. Since these applications solely rely on the *semantic property*, and not on its approximation by the abstract interpretation, we can independently prove the correctness of the applications. Therefore, providing a different abstract interpretation or other method for obtaining information on the escape behaviour of programs will not invalidate the correctness results of this chapter.

Compile–time garbage collection detects positions in a program where parts of data structures will become garbage during execution, and modifies the program to cater for these situations. This includes both the deletion of constructor nodes and closure chains. For closures, we additionally can perform *efficient closure utilisation* to avoid heap allocation of closures at all.

In addition to the proofs of correctness of these applications, we study the effects of the applications. The experiments show that the runtime behaviour is improved.

7.1 Compile–Time Garbage Collection

Several methods have been proposed to reduce the runtime memory consumption of functional programs by compile–time garbage collection (ctgc). The approaches differ in the way how the memory management strategy is altered by the information obtained. Especially, the *location of memory reuse* and the *way of memory reuse* is of importance.

The location of memory reuse determines at which point of execution a garbage cell is reused. Here, we distinguish two approaches:

Immediate reuse: This approach has the advantage of keeping the number of garbage cells small at the price of frequent interruptions of the actual computation. It must be guaranteed that sharing of the argument being reused does not occur. Therefore, either (1) the (implementation of the) function must be altered to receive additional arguments indicating an "unshared situation", or (2) specially trimmed versions of the function must be used in the appropriate situations. Both approaches have major deficiencies: (1) requires additional tests within the function, which increase the time spent on memory management operations; (2) avoids this but can increase the code size exponentially in the number of arguments of the function.

Delayed reuse: Deallocations are performed at some later point in execution, maybe even the end of the corresponding function call. The advantage is that more deallocations are performed at the same time, which may be done more efficiently. Also, there is no need for modifications within the function, which circumvents the above problems.

The *way of memory reuse* can be either:

Deallocating: This is done by adding the cell to the free part of the heap.

Direct reuse: It can be used in situations where the deallocation is immediately followed by an allocation, resulting in an "in-place" version of the function.

Using the above characteristics, our approach can be described as *delayed deallocating*: We insert deallocation annotations into the programs, which cause the deallocation of heap cells after the evaluation of an application. The deleted cells were created during the evaluation of the arguments of the application. We circumvent the problems with shared structures by considering only those locations which were created during the evaluation of the argument of an application. Such cells cannot be shared.

7.1.1 Program Annotations

We extend the sets of expressions and programs as defined in Definition 2.4 by allowing one additional construct; an *annotated version of saturated application*. The general scheme is to annotate the argument e_i in applications $(e\ e_1\ \ldots\ e_m)$, where parts of e_i do *not* escape from the (evaluation of the) expression.

Definition 7.1 (Annotations)
We define the *family of annotations* $\aleph := \langle \aleph^t \mid t \in \mathsf{T}(S) \rangle$, where the sets of annotations of type t are defined by induction on t:

$t = bs \in BS$: $\aleph^{bs} := \mathbb{B}$

$t = cs \in CS$: $\aleph^{cs} := \mathfrak{P}\left(\bigcup_{cs \overset{\star}{\twoheadleftarrow} cs'} \{(c, i) \mid c \in C^{t_1,\ldots,t_n \to cs'}, t_i \in BS, 0 \leq i \leq n\}\right)$

$t = t_1, \ldots, t_m \to t_0$: $\aleph^{t_1,\ldots,t_n \to t} := \mathbb{B} \times \mathbb{B}$ $\triangleleft$

Remarks:

- For basic types the annotations are either 0 or 1, indicating whether all cells allocated for this result are to be deleted.
- For constructed sorts, an annotation is a set of pairs (c, i), where c is a constructor and i is either 0 or the index of an argument of c with basic sort as type. More precisely, the constructor c must be one of those constructors which may occur as the top constructor of one of the subterms of a value of type cs. Formally, we describe this by using the notation $cs \overset{\star}{\twoheadleftarrow} cs'$, which was introduced in Chapter 4.
- For functional types the annotations are pairs; each component is either 0 or 1. The first component determines whether the chain of closures is to be deleted, and the second component affects the parameter bindings in the closures. In analogy to the `ListOfInt` type, the chain of closures is the spine, and the parameters are the entries.

The semantics of the annotations is formally defined in Definitions 7.4 and 7.5.

Definition 7.2 (CTGC–Annotated Expressions, Programs)
We define the family of all *ctgc–annotated expressions* as $E_c := \langle E_c^t \mid t \in \mathsf{T}(S) \rangle$, where the sets E_c^t are defined as:

1. $E^t \subseteq E^t_c$
2. $(e\ e_1^{a_1}\ \dots\ e_n^{a_n}) \in E^t_c$ if $e \in E_c^{t_1,\dots,t_n \to t}$, $e_i \in E^t_c$, and $a_i \in \aleph^{t_i}$ $(1 \le i \le n)$

An *ctgc–annotated* F*-program* is a finite set of definitions where the right hand sides are annotated expressions. $\lhd$

To provide a semantics for annotated expressions, we define a function which performs deallocations in graphs.

Definition 7.3 (Graph Deletion)
Let $G = (g, L) \in \mathsf{G}$ be a graph and let $l \in \mathsf{Dom}(g)$ be a location. The graph resulting from *deletion of the node* $g(l)$ *in* G is defined as $G - l := (g', L')$ where $\mathsf{Dom}(g') = \mathsf{Dom}(g) \setminus \{l\}$ and $L' = L \setminus \{l\}$. $\lhd$

Remarks:

- This notion is the counterpart to the graph allocation function $G + hn$. More precisely, if we have $(l', G') = G + hn$ then $G' - l' = G$.
- The order of deletions is not important, i.e. $(G - l_1) - l_2 = (G - l_2) - l_1$. Therefore, we can introduce the following abbreviation: $G - \{l_1, \dots, l_n\} := (\dots(G - l_1)\dots) - l_n$.

Lemma 7.1 (Deletion of Garbage Preserves Order)
Let $G = (g, L) \in \mathsf{G}$ be a graph and let $l \in \mathsf{Dom}(g)$ be a location. We have $l \notin L$ iff $G - l \preccurlyeq G$ and $G \preccurlyeq G - l$. $\Box$

Proof Trivial. q.e.d.

This property is the key to the design of the graph domain. With a graph order which does not ignore garbage cells, the deletion of graph cells would be non–monotonic.

We use this function to define the deallocation of more than one graph node corresponding to an annotation.

Definition 7.4 (Annotated Graph Deletion)
Let $G \in \mathsf{G}$ be a graph, $a \in \aleph^t$ be an annotation, and $l_0 \in \mathsf{Dom}(g^t)$, $L \subseteq \overrightarrow{l_0}$ be locations. The graph resulting from *deletion of the nodes in* $\overrightarrow{l_0}$ *restricted to* L *in* G *corresponding to the annotation* a is defined as $G -_{a/L} l_0 := G_r - N$, where the graph G_r and the set N are defined in the following way:

1. $G_r := G$ and $N := \begin{cases} \{l_0\} & \text{if } a = 1 \\ \emptyset & \text{otherwise} \end{cases}$ if $t = bs \in BS$.
2. $G_r := G_n$ and the graphs G_i and the set N are defined as

$$\begin{array}{rcll} G_0 & := & G & \\ G_i & := & \begin{cases} G_{i-1} -_{a/L_i} l_i & \text{if } t'_i \in CS,\ L_i := \overrightarrow{l_i} \cap L \\ G_{i-1} & \text{if } t'_i \in BS \end{cases} & 1 \le i \le n \\ N & := & \{l_i \mid 0 \le i \le n,\ (c, i) \in a\} & \end{array}$$

if $t = cs \in CS$ and $g(l_0) = \langle c, l_1, \dots, l_n \rangle$ with $c \in C^{t_1,\dots,t_n \to t}$.

3. $G_r := G$ and $N := \begin{cases} \{l_0\} & \text{if } a = (1, b_2) \\ \emptyset & \text{otherwise} \end{cases}$ if $t = t_1, \ldots, t_n \to t_0 \in \mathsf{T}(S)$, $g(l_0) = \langle \varphi \rangle$.
4. $G_r := G -_{a/L_f} l_f$, where $L_f := L \cap \overleftrightarrow{l_f}$ and $N := \{l_0 \mid a = (1, b_2)\} \cup \{L \cap \overleftrightarrow{l_i} \mid 1 \le i \le n \wedge a = (b_1, 1)\}$ if $t = t_1, \ldots, t_n \to t_0 \in \mathsf{T}(S)$, $g(l_0) = \langle @, l_f, l_1, \ldots, l_n \rangle$. $\lhd$

The graph component spanned by l_0 is the part in which deletion is performed. The set L restricts the set of cells of $\overleftrightarrow{l_0}$ which are to be deleted. We need this set to avoid that cells are deleted which were already present before the evaluation; these cells can be shared which means that they are active and hence their deletion would be incorrect.

Remarks:

- For basic types, the annotation a determines whether the heap cell is deleted ($a = 1$) or not ($a = 0$).
- For constructed sorts, the deallocation consists of three phases:
  1. Recursive deletion corresponding to the annotation a in all argument components l_i which have a constructed type; this is done by successive creation of the graphs G_i.
  2. Delete the top cell $g(l_0) = \langle c, l_1, \ldots, l_n \rangle$, if the tuple $(c, 0)$ is an element of the annotations.
  3. Furthermore, delete those l_i where (c, i) is element of the annotations. By definition, this can only occur for arguments which have a basic sort as type. This phase is necessary to remove boxed basic values.

  Because the order of the deletions is not important, it does not matter in which order these steps are performed either.
- For functional types, we perform similar steps: For an annotation (b_1, b_2), we delete the spine of the closure chain if $b_1 = 1$, and the entries in the closure chain if $b_2 = 1$. However, in contrast to constructed types, there is no distinction between different entry types. We cannot do this here, because there is no possibility to determine at compile time which parameter profiles are present in the closure chain.

Example: Consider the graph function g_{demo} given in Figure 6.2 on page 65. We choose $G_{\text{demo}} := (g_{\text{demo}}, \mathsf{Dom}(g_{\text{demo}}))$.

- If we consider $G_{\text{demo}} -_{l_3/\{l_3,l_4,l_5\}} \{(\texttt{Cons}, 0), (\texttt{Cons}, 1)\}$, we obtain a graph were l_3 is deleted because of the annotation $(\texttt{Cons}, 0)$ and l_4 is deleted because of $(\texttt{Cons}, 1)$. On the other hand, l_5 remains, since there is no annotation $(\texttt{Nil}, 0)$.
- For the annotation (b_1, b_2) and the location l_1, we delete l_1 and l_2 if $b_1 = 1$, and l_3, l_4, l_5 if $b_2 = 1$. $\diamond$

With this auxiliary function, we can now give the definition of the graph semantics of ctgc-annotated expressions in extension of Definition 6.13.

Definition 7.5 (Graph Semantics $\mathfrak{G}_c$ of CTGC-Annotated Expressions)
Let $e \in E_c^t$ be a ctgc–annotated expression, $G \in \mathsf{G}$ be a graph, $\xi \in \mathsf{Env}(X, \mathsf{P})$ be an environment for variables, and $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ be an environment for defined functions. The *graph semantics of e* $\mathfrak{G}_c[\![e]\!](\xi, \theta, G) \in \mathsf{P}^t \times \mathsf{G}$ is defined inductively on the structure of e in analogy to Definition 6.13, with the following extension:

- $\mathfrak{G}_c[\![(e\ e_1^{a_1}\ \dots\ e_n^{a_n})]\!](\xi, \theta, (g, L)) := \mathsf{mkgarb}(L, (p_r, G_r)) -_{a_1/\Delta_1} p_1 -_{a_2/\Delta_2} \cdots -_{a_n/\Delta_n} p_n$

$$\begin{array}{rrcl} \text{where} & (p_0, (g_0, L_0)) & := & \mathfrak{G}_c[\![e_0]\!](\xi, \theta, (g, L)) \\ & (p_j, (g_j, L_j)) & := & \mathfrak{G}_c[\![e_j]\!](\xi, \theta, (g_{j-1}, L_{j-1})) \text{ for } 1 \leq j \leq m \\ & (p_r, G_r) & := & \mathsf{exec}^{t_1, \dots, t_m \to t}(\theta, p_0, (g_m, L_{\underline{m}}))(p_1, \dots, p_m, (g_m, L_m)) \\ & \Delta_i & := & (\mathsf{Dom}(g_i) \setminus \mathsf{Dom}(g_{i-1})) \cap \overrightarrow{p_i} \text{ for } 1 \leq i \leq n \end{array}$$

for $e_0 \in E_c^{t_1, \dots, t_n \to t_0}$, $e_i \in E_c^{t_i}$, and $a_i \in \aleph^{t_i}$ $(1 \leq i \leq m)$. $\lhd$

An annotated application $(e\ e_1^{a_1}\ \dots\ e_n^{a_n})$ is interpreted in the following way: While evaluating the expression $(e\ e_1\ \dots\ e_n)$ all allocations in the graph are recorded such that the sets Δ_i contain those locations which were allocated during the evaluation of e_i and are part of the result. After performing the function call and executing mkgarb, the locations in Δ_i are deleted corresponding to the annotation a_i.

Example: Consider the annotated expression (`qs (filter` $(<$ `a) l)`$^{\{(\texttt{Cons},0),(\texttt{Nil},0)\}}$) where `filter` and `qs` are defined in the program P

```
filter f []     = []
filter f (x:xs) = if (f x) (f:(filter f fs)) else (filter f fs)
qs []    = []
qs (a:l) = (qs (filter (<a) l))++[a]++(qs (filter (>=a) l))
```

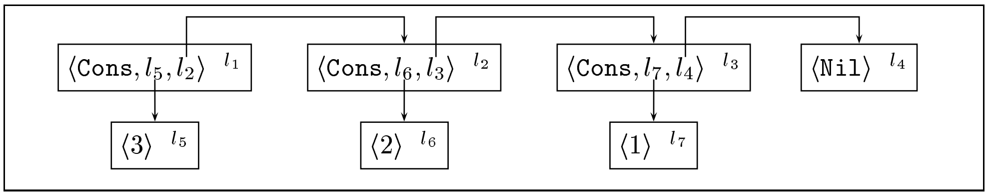
Furthermore, we use bindings $\theta = \mathfrak{G}[\![P]\!] = [\texttt{filter}/\mathit{filter},\ \texttt{qs}/\mathit{qs}]$, $\xi = [\texttt{l}/l_1,\ \texttt{a}/l_5]$, and the graph $(g, \mathsf{Dom}(g))$ where the graph function g is shown in Figure 7.1(a).

Evaluating the arguments of the top–level application yields the results $(l_8, (g_0, L_0)) = \mathfrak{G}_c[\![\texttt{qs}]\!](\xi, \theta, (g, L))$ and $(l_9, (g_1, L_1)) = \mathfrak{G}_c[\![\texttt{(filter (<a) l)}]\!](\xi, \theta, (g_0, L_0))$. The resulting graph g_1 is shown in Figure 7.1(b)[1]. Hence, we have $\Delta_1 = \{l_9, l_{10}, l_{11}\}$. Note that l_8 is allocated during the evaluation of `qs` and therefore is already present in g_0.

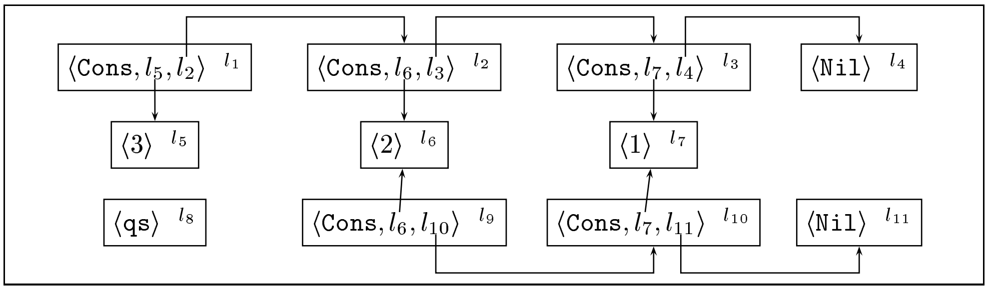
Immediately after the call to `qs` and the evaluation of mkgarb, we have the graph (g_2, L_2) shown in Figure 7.1(c), where garbage cells are set with a dashed border[2].

Finally, the evaluation of $(l_{12}, (g_2, L_2)) -_{\{(\texttt{Cons},0),(\texttt{Nil},0)\}/\{l_9, l_{10}, l_{11}\}} l_9$ removes almost all garbage cells, with the exception of l_8. The graph (g_r, L_r) which results from evaluating the annotated expression (`qs (filter` $(<$ `a) l)`$^{\{(\texttt{Cons},0),(\texttt{Nil},0)\}}$) is shown in Figure 7.1(d). $\diamond$

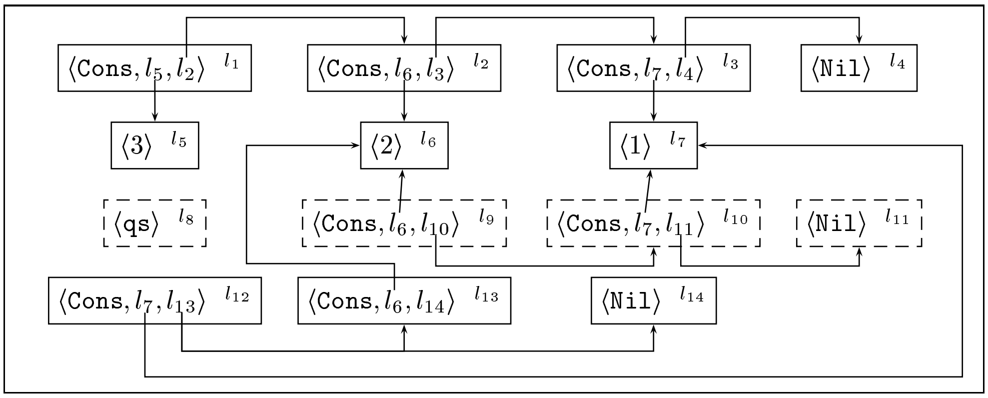
[1] For simplicity, we omitted the garbage nodes allocated by the evaluation of `(filter (<a) l)`.
[2] As before, we omitted garbage cells created while evaluating the call to `qs`.



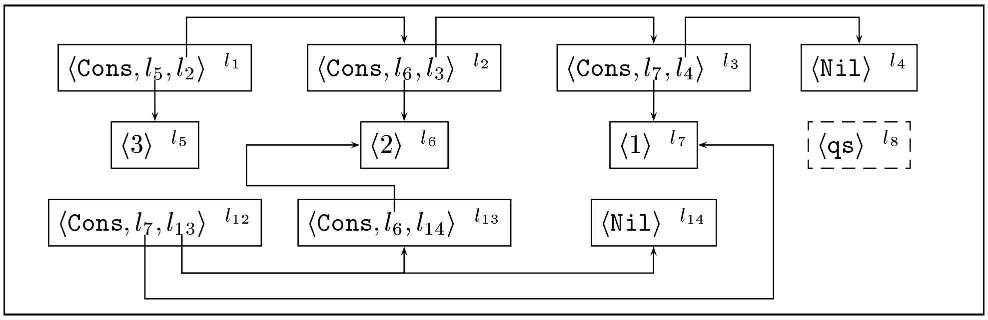
(a) g: Start



(b) g_1: Evaluated Top–Level Arguments



(c) g_2: After `qs`



(d) g_r: Result

Fig. 7.1: Evolution of the Example Graph

7.1.2 Correctness

Of course, arbitrary annotations do not preserve the meaning of expressions. This is guaranteed only if deletions affect only garbage locations. To ensure this, it is sufficient that no cell from Δ_i which is reachable from the result is deleted. Hence, non–escaping is sufficient to preserve the semantics. This is formalised by the notion of well–annotated expressions.

Definition 7.6 (Well-Annotated Expression)
A ctgc–annotated expression $e_0 \in E_c$ of a program P is called *well–annotated* iff for all $e \in E_c^{t_1,\ldots,t_n \to t}$, $e_i \in E_c^t$, and $a_i \in \aleph^{t_i}$ $(1 \leq i \leq n)$ holds that if $(e\ e_1^{a_1}\ \ldots\ e_n^{a_n})$ is a sub expressions of e then all parts of e_i which escape from $(e\ e_1\ \ldots\ e_n)$ are not deleted by $\mathfrak{G}_c$. $\lhd$

We show that well–annotated expression have the same meaning as their non–annotated counterparts. To clarify what “meaning” is in this context, we have to consider that we do not want the resulting graphs to be equal. Instead, we use the representation function introduced in the last chapter to establish a connection between annotated and non–annotated expressions.

Corollary 7.1 (Well–Annotated Expressions are Monotonic)
Let $e \in E_c^t$ be a well–annotated expression, $G, G' \in \mathsf{G}$ be graphs such that $G \preccurlyeq G'$, $\xi \in \mathsf{Env}(X, \mathsf{P})$ be an environment for variables, and $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ be an environment for defined functions. Given that, we have $\mathfrak{G}_c[\![e]\!](\beta, \sigma, G) \preccurlyeq_{\mathsf{P}^t \times \mathsf{G}} \mathfrak{G}_c[\![e]\!](\beta, \sigma, G')$. □

This result guarantees that we can define a fixpoint semantics for ctgc–annotated programs based on the expression semantics for annotated expressions. This is only possible because of our model of graphs in conjunction with the generalised fixpoint Theorem 6.2 from the previous chapter.

Theorem 7.1 (Correctness Of Compile–Time Garbage Collection)
Let $e \in E_c^t$ be a well–annotated expression, $G \in \mathsf{G}$ be a graph, $\xi \in \mathsf{Env}(X, \mathsf{P})$ be an environment for variables, and $\theta \in \mathsf{Env}(DF, \mathsf{GOps})$ be an environment for defined functions.

1. If $(l_r, G_r)' = \mathfrak{G}_c[\![e]\!](\beta, \sigma, G)$ then we have $G \preccurlyeq G_r$.
2. Let $e' \in E^t$ be the expression resulting from e by removing all annotations, and let $(l, (g, L)) = \mathfrak{G}_c[\![e]\!](\beta, \sigma, G)$ and $(l', (g', L')) = \mathfrak{G}[\![e']\!](\beta, \sigma, G)$. Then we have
$$\mathsf{rep}_g(l, \sigma) = \mathsf{rep}_{g'}(l', \sigma)$$ □

The first item ensures that no “old” cell is deleted by the evaluation, i.e. no allocated cell is deleted during the evaluation. The second item establishes the actual correctness of the result.

Proof

1. During evaluation of annotated application, only locations in the sets Δ_i are deleted. These sets contain only locations allocated during evaluation of arguments. Hence, no locations already present before evaluating the i–th argument is deleted.

2. The only chance to violate the equation (cf. Lemma 7.1)is that a heap cell contributing to the result is deleted in the evaluation of the expression e. This means that we have an $l \in \Delta_i$ which is deleted by $G' -_{a_i/\Delta_i} p_i$ and is not garbage, i.e. $l \in \overrightarrow{p_r}$. However, this means that l escapes from the application which is a contradiction to our assumption. q.e.d.

7.1.3 Using the Abstract Interpretation $\mathfrak{E}$ for Finding Annotations

Following our model–based approach, we considered the correctness of ctgc solely based on the semantical property. Hence, the results are valid independently of the means used to determine escaping. Especially, the last result does not depend on any special property of our abstract interpretation $\mathfrak{E}$, not even on the choice of the abstract domains.

In this subsection, we demonstrate how we can use the abstract interpretation $\mathfrak{E}$ for finding annotations. We start by defining a function $\mathsf{abstoann}$ which determines the annotations corresponding to an abstract value.

Definition 7.7 (Abstract Values to Annotations)
The family of functions $\mathsf{abstoann} = \langle \mathsf{abstoann}^t \mid t \in \mathsf{T}(S)\rangle$, where $\mathsf{abstoann}^t : \mathsf{A}^t \to \aleph^t$, is defined as induction on t:

$t = bs \in BS$: $\mathsf{abstoann}^{bs}(b) := b$ for all $b \in \mathsf{A}^{bs} = \mathbb{B} = \aleph^{bs}$.

$t = cs \in CS$: $\mathsf{abstoann}^{cs}((b,a)) := C_b \cup R_a \cup N_a$ where the sets are defined as

$$\begin{aligned}
C_b &:= \begin{cases} \{(c,0) \mid c \in C^{t_1,\dots,t_n \to cs'},\ cs' \in [cs]\} & \text{if } b = 1 \\ \emptyset & \text{otherwise} \end{cases} \\
R_a &:= \bigcup_{cs' \in [cs], c \in C^{t_1,\dots,t_n \to cs'}, [t_i] \neq [cs], t_i \in CS} \mathsf{abstoann}^{t_i}(a_{cs',c,t_i}) \\
N_a &:= \bigcup_{cs' \in [cs], c \in C^{t_1,\dots,t_n \to cs'}, [t_i] \neq [cs], t_i \in BS} \{(c,i) \mid \mathsf{abstoann}^{t_i}(a_{cs',c,t_i}) = 1\}
\end{aligned}$$

for all $(b,a) \in \mathsf{A}^{cs}$, i.e. $a = \prod_{cs' \in [cs]} \prod_{c \in C^{t_1,\dots,t_n \to cs'}} \prod_{[t_i] \neq [cs]} a_{cs',c,t_i}$.

$$t = t_1,\dots,t_m \to t_0:\quad \mathsf{abstoann}^{t_1,\dots,t_n \to t}((b,f)) := \begin{cases} (1,1) & \text{if } b = 1 \text{ and } f(\perp^{t_1},\dots,\perp^{t_n}) = \perp^t \\ (1,0) & \text{if } b = 1 \text{ and } f(\perp^{t_1},\dots,\perp^{t_n}) \neq \perp^t \\ (0,0) & \text{otherwise} \end{cases}$$

for all $(b,f) \in \mathsf{A}^{t_1,\dots,t_n \to t}$. $\lhd$

In this context, an abstract value is interpreted as indicator for the escape behaviour: Non–zero bits indicate that the corresponding part does not escape:

- For basic types, the choice is obvious.
- In case of constructed types, the resulting annotation consists of three components: the set C_b which determines how the constructors of $[cs]$ are handled, the set R_a, which affects all other reachable constructors, and the set N_a, which determines how the constructor arguments of basic type are handled.

- Special consideration is necessary for functional types; the escape tag b determines whether the closure escapes. If it does not, the spine of the closure chain is deleted. Furthermore, the entries are deleted if the functional component f indicates that there is no escaping of entries stored in the closure chain.

Examples:

1. For the type $t = \texttt{ListOfInt}$, we obtain the following values:

$$\begin{array}{rcl}\mathsf{abstoann}^{\texttt{ListOfInt}}(0,0) & = & \emptyset \\ \mathsf{abstoann}^{\texttt{ListOfInt}}(1,0) & = & \{(\texttt{Cons},0),(\texttt{Nil},0)\} \\ \mathsf{abstoann}^{\texttt{ListOfInt}}(0,1) & = & \{(\texttt{Cons},1\} \\ \mathsf{abstoann}^{\texttt{ListOfInt}}(1,1) & = & \{(\texttt{Cons},0),(\texttt{Nil},0),(\texttt{Cons},1)\}\end{array}$$

   We can see that it is sufficient to compute $\mathsf{abstoann}^{cs}$ for the values with exactly one non–zero entry. Furthermore, we have that $\mathsf{abstoann}^{cs}(a) = \emptyset$ iff $a = \bot^{cs}$.

2. Consider again the $\texttt{ITree}/\texttt{CTree}$ example from Section 4.1. Here we have:

$$\begin{array}{rcl}\mathsf{abstoann}^{\texttt{ITree}}(1,((0,0),(0,0))) & = & \{(\texttt{ILeaf},0),(\texttt{INode},0),(\texttt{CLeaf},0),(\texttt{CNode},0)\} \\ \mathsf{abstoann}^{\texttt{ITree}}(0,((1,0),(0,0))) & = & \{(\texttt{ILeaf},1)\} \\ \mathsf{abstoann}^{\texttt{ITree}}(0,((0,1),(0,0))) & = & \{(\texttt{INode},1)\} \\ \mathsf{abstoann}^{\texttt{ITree}}(0,((0,0),(1,0))) & = & \{(\texttt{CLeaf},1)\} \\ \mathsf{abstoann}^{\texttt{ITree}}(0,((0,0),(0,1))) & = & \{(\texttt{CNode},1)\}\end{array}$$

3. For the higher–order type $t = \texttt{int} \to \texttt{int}$, we obtain:

$$\begin{array}{rcl}\mathsf{abstoann}^{\texttt{int}\to\texttt{int}}(0,[0 \mapsto 0,\ 1 \mapsto y]) & = & (0,0) \\ \mathsf{abstoann}^{\texttt{int}\to\texttt{int}}(1,[0 \mapsto 0,\ 1 \mapsto y]) & = & (1,1) \\ \mathsf{abstoann}^{\texttt{int}\to\texttt{int}}(b,[0 \mapsto 1,\ 1 \mapsto y]) & = & (b,0)\end{array}$$

As before we handle functional abstract values in a slightly different way than non–functional values. In Section 5.3, we introduced the notion of void abstract values because functional values contain the functions escape behaviour in addition to the escape behaviour of the closure. For the algorithm which annotates an expression, we have to tag the escape behaviour of the closure without changing the functions escape behaviour. Therefore, we introduce a function which compares the functional behaviour of abstract values.

Definition 7.8 (Functional Equivalence)
Let $t \in \mathsf{T}(S)$ be a type. Two abstract values $a_1, a_2 \in \mathsf{A}^t$ are *functionally equivalent* ($a_1 \simeq_f a_2$) iff either $t \in S$ or $t = t_1, \ldots, t_n \to t_0$ and $a_i = (b_i, f)$ for $b_i \in \mathbb{B}$ and $f : \mathsf{A}^{t_1} \times \cdots \times \mathsf{A}^{t_n} \to \mathsf{A}^t$. $\lhd$

Now we describe an algorithm which annotates an expression using the results of the abstract interpretation $\mathfrak{E}$.

Definition 7.9 (CTGC–Annotation of Expressions Based on $\mathfrak{E}$)
The family of functions $\mathsf{ctgc}_{\mathfrak{E}} = \langle \mathsf{ctgc}^t_{\mathfrak{E}} \mid t \in \mathsf{T}(S)\rangle$, where $\mathsf{ctgc}^t_{\mathfrak{E}} : E^t \times \mathsf{Env}(X, \mathsf{A}) \times \mathsf{Env}(DF, \mathsf{AOps}) \to E^t_c$, is defined by induction on the structure of expressions.

- $\mathsf{ctgc}^t_{\mathfrak{E}}[\![\varphi]\!](\chi, \varphi) := \varphi$ for $\varphi \in X^t \cup DF^t \cup \Omega^t$

- $\mathsf{ctgc}^t_{\mathfrak{E}}[\![(e_0\ e_1\ \ldots\ e_m)]\!](\chi, \varphi) := (e'_0\ e'_1\ \ldots\ e'_m)$ for $e_i \in E^{t_i}$ $(0 \le i \le m)$, $t_0 = t_1, \ldots, t_n \to t_r$, $m < n$ where $e'_i = \mathsf{ctgc}^{t_i}_{\mathfrak{E}}[\![e_i]\!](\chi, \varphi)$ $(0 \le i \le m)$

- $\mathsf{ctgc}^t_{\mathfrak{E}}[\![(e_0\ e_1\ \ldots\ e_n)]\!](\chi, \varphi) := (e'_0\ {e'_1}^{a_1}\ \ldots\ {e'_n}^{a_n})$ for $e_i \in E^{t_i}$ $(0 \le i \le m)$, $t_0 = t_1, \ldots, t_n \to t_r$ where $e'_i = \mathsf{ctgc}^{t_i}_{\mathfrak{E}}[\![e_i]\!](\chi, \varphi)$ $(0 \le i \le n)$, $a_i = \mathsf{abstoann}^{t_i}(x_i)$ $(1 \le i \le m)$ where $(x_1, \ldots, x_n) \in \mathsf{A}^{t_1} \times \cdots \times \mathsf{A}^{t_n}$ are maximal abstract values with $x_i \simeq_f \mathfrak{E}[\![e_i]\!](\chi, \varphi)$ such that if $(b, f) = \mathfrak{E}[\![e_0]\!](\chi, \varphi)$ then $f(x_1, \ldots, x_n) = \bot^t$.

- $\mathsf{ctgc}^t_{\mathfrak{E}}[\![\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](\chi, \varphi) := \texttt{if}\ e'_0\ \texttt{then}\ e'_1\ \texttt{else}\ e'_2$ for $e_i \in E^{t_i}$ $(0 \le i \le 2)$, $t_0 = \texttt{bool}$ where $e'_i = \mathsf{ctgc}^{t_i}_{\mathfrak{E}}[\![e_i]\!](\chi, \varphi)$ $(0 \le i \le 2)$ $\lhd$

Remarks:

- The original expression is traversed recursively, leaving every construct except saturated applications unchanged.

- For saturated applications, we search for maximal abstract values without escaping. To preserve computations influenced by functional parameters, we restrict the functional component of the abstract values to those which coincide with the abstract function determined by $\mathfrak{E}$. For this purpose we use the notion of functional equivalence. If we would omit it, the annotation would still be correct, but would be of lower quality.

- To guarantee that we can choose uniquely determined maximal values in the case of saturated applications, we use the additivity of $\mathfrak{E}$ (Theorem 4.1). Assume that we have two tuples of maximal values $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ such that $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) = \bot^t$. By Theorem 4.1 we have

  $$f((x_1, \ldots, x_n) \sqcup^{t_1 \times \cdots \times t_n} (y_1, \ldots, y_n)) = f(x_1, \ldots, x_n) \sqcup^t f(y_1, \ldots, y_n) = \bot^t$$

  Because both tuples are maximal, we then also have

  $$(x_1, \ldots, x_n) = (y_1, \ldots, y_n) = (x_1, \ldots, x_n) \sqcup^{t_1 \times \cdots \times t_n} (y_1, \ldots, y_n)$$

Example: The following program P is a complete implementation of the quicksort algorithm. It consist of three definitions, which are notated in Haskell syntax.

```
[] ++ x = []
(a:l) ++ x = (a:l++x)

filter p [] = []
filter p (a:l) = if (p a) then a:(filter p l) else (filter p l)

qs [] = []
qs (a:l) = (qs (filter (<a) l))++([a]++(qs (filter (>=a) l)))
```

When we try to annotate the right hand sides of the function definitions using a variable environment χ which assigns to all occurring variables the worst–case behaviour and the function environment $\mathfrak{C}[\![P]\!]$, we observe that only the second definition of `qs` changes; all other right hand sides become annotated with trivial annotations without effect. For the second definition of `qs`, we obtain the following annotations with $\chi = [\texttt{a}/1,\ \texttt{l}/(1,1)]$:

$$\begin{array}{ll}
& \mathsf{ctgc}^{\texttt{ListOfInt}}_{\mathfrak{C}}[\![\texttt{(qs (filter (<a) l))++([a]++(qs (filter (>=a) l)))}]\!](\chi, \mathfrak{C}[\![P]\!]) \\
= & \quad \texttt{(qs (filter (<a)}^{1:(1,1)}\ \texttt{l}^{2:\{(\texttt{Cons},0),(\texttt{Nil},0)\}}\texttt{)}^{3:\{(\texttt{Cons},0),(\texttt{Nil},0)\}}\texttt{)}^{4:\{(\texttt{Cons},0),(\texttt{Nil},0)\}} \\
& \texttt{++(}\quad \texttt{[a]}^{5:\{(\texttt{Cons},0),(\texttt{Nil},0)\}} \\
& \qquad \texttt{++(qs (filter (>=a)}^{6:(1,1)}\ \texttt{l}^{7:\{(\texttt{Cons},0),(\texttt{Nil},0)\}}\texttt{)}^{8:\{(\texttt{Cons},0),(\texttt{Nil},0)\}}\texttt{))}
\end{array}$$

To simplify the explanation, the annotations are preceded by numerical labels:

1: and 2: annotate the arguments of the application `(filter (<a) l)`.

We have $\mathfrak{C}[\![\texttt{(<a)}]\!](\chi, \mathfrak{C}[\![P]\!]) = (0, [0 \mapsto 0,\ 1 \mapsto 0])$ and $\mathfrak{C}[\![\texttt{l}]\!](\chi, \mathfrak{C}[\![P]\!]) = (1,1)$. However, only the abstract value of the partial application is of interest, because the notion of functional equivalence affects only values of functional type. This is because we have to ensure that the functional behaviour is preserved while searching maximal abstract values a_1 and a_2 such that $(\mathfrak{C}[\![P]\!](\texttt{filter}))(a_1, a_2) = \bot^{\texttt{ListOfInt}}$. Hence, we have $a_1 \in \{(0, [0 \mapsto 0,\ 1 \mapsto 0]),\ (1, [0 \mapsto 0,\ 1 \mapsto 0])\}$. Corresponding to Table 7.1, where we find the values for $\mathfrak{C}[\![P]\!](\texttt{filter})$ for these possibilities, we have $a_1 = (1, [0 \mapsto 0,\ 1 \mapsto 0])$ and $a_2 = (1, 0)$. Application of $\mathsf{abstoann}$ yields the annotations.

a_1	a_2	$(\mathfrak{C}[\![P]\!](\texttt{filter}))(a_1, a_2)$
$(0, [0 \mapsto 0,\ 1 \mapsto 0])$	$(0,0)$	$(0,0)$
$(0, [0 \mapsto 0,\ 1 \mapsto 0])$	$(0,1)$	$(0,1)$
$(0, [0 \mapsto 0,\ 1 \mapsto 0])$	$(1,0)$	$(0,0)$
$(0, [0 \mapsto 0,\ 1 \mapsto 0])$	$(1,1)$	$(0,1)$
$(1, [0 \mapsto 0,\ 1 \mapsto 0])$	$(0,0)$	$(0,0)$
$(1, [0 \mapsto 0,\ 1 \mapsto 0])$	$(0,1)$	$(0,1)$
$(1, [0 \mapsto 0,\ 1 \mapsto 0])$	$(1,0)$	$(0,0)$ *
$(1, [0 \mapsto 0,\ 1 \mapsto 0])$	$(1,1)$	$(0,1)$

Tab. 7.1: $\mathfrak{C}[\![P]\!](\texttt{filter})$

During execution, only the annotation (1) will have an effect: It removes the closure for `(<a)`. However, since the cell representing `a` is already present before the evaluation of the argument, nothing is removed there. For the same reason, annotation (2) has no effect at all. In general, no annotation of variables will have an effect.

3: annotates the argument of the first recursive call to `qs`.

The maximal value a such that $(\mathfrak{C}[\![P]\!](\texttt{qs}))(a) = \bot^{\texttt{ListOfInt}}$ is $a = (1, 0)$, i.e. the constructors of the argument list do not escape from `qs`. Hence, we have the annotation $\mathsf{abstoann}^{\texttt{ListOfInt}}((1,0)) = \{(\texttt{Cons}, 0), (\texttt{Nil}, 0)\}$.

4: annotates the result of the first recursive call to `qs`, which is the first argument to the outer call to append (`++`).

The constructors of the first argument of `++` do not escape and hence the annotation is $\{(\texttt{Cons}, 0), (\texttt{Nil}, 0)\}$.

5: annotates the expression `[a]`, which is the first argument of the inner call to `++`.

Here we have a special situation: The two constructor cells allocated during evaluation of the expression are immediately deallocated after `++`. Typically, this does not occur for constructor cells but more often for closures. In a stack–based implementation of graph reduction, we could allocate such cells in the stack frame of the incarnation of `qs` instead of allocating them in the heap. We consider these cases in Section 7.2.

6: and 7: analogous to *1: and 2:*

8: analogous to *3:* $\diamond$

We show that the annotated expressions we obtain by using the function $\mathsf{ctgc}_{\mathfrak{E}}$ are well–annotated, if the environments are chosen in the way described in the example.

Theorem 7.2 ($\mathsf{ctgc}_{\mathfrak{E}}$ Yields Well–Annotated Results)
Let $e_0 \in E$ be an expression of a program P, X be the variables occurring in e, and $\chi \in \mathsf{Env}(X, \mathsf{A})$ be an environment such that $\chi(x) = \top^t$. Then we have that $\mathsf{ctgc}^t_{\mathfrak{E}}[\![e]\!](\chi, \mathfrak{E}[\![P]\!])$ is well–annotated. $\square$

Proof Without loss of generality we can assume that $e = (e_0\ e_1\ \ldots\ e_n)$ is a saturated expression. The rest follows immediately by induction.

Corresponding to Definition 7.6 we have to prove that for all argument expressions e_i $(1 \leq i \leq n)$ holds that all parts of e_i which escape from e are not deleted by $\mathfrak{G}_c$. Therefore, it is sufficient that no part which escapes is annotated. We know with Lemma 5.5 that $\mathfrak{E}$ approximates escaping if the environments are chosen correctly. By definition, χ and $\mathfrak{E}[\![P]\!]$ have this property and hence if $(b, f) = \mathfrak{E}[\![e_0]\!](\chi, \mathfrak{E}[\![P]\!])$ then $f(x_1, \ldots, x_n) = \perp^t$ is a sufficient condition to ensure that no part which escapes is annotated. q.e.d.

7.1.4 Experimental Results

We have done experiments with programs which were annotated by $\mathsf{ctgc}_{\mathfrak{E}}$. Instead of using an abstract machine as execution basis of graph reduction, the programs were compiled by hand from our example language to C. This has several advantages:

1. We can directly translate the graph reduction process described in Chapter 6 by using C's runtime system. The recursive structure of the program can be maintained, using C's runtime heap as implicit argument of all functions. A detailed description of this translation can be found in [Moh95c, Moh95d].
2. The runtime and memory behaviour can easily be traced by insertion of appropriate statements.

3. The measurements focus on the effects caused by ctgc. In highly optimised implementations of a realistic programming language, several optimisations interfere. As a consequence, their mutual effects may hinder or intensify each other. Therefore, it can be hard to rate the results of a single optimisation.

The Quicksort Program

The first example is the `qs` program. Because the memory consumption is mainly determined by the list cells, the higher–order function `filter` was replaced by first–order variants. The complete C source can be found in Appendix C.

The first experiment was to measure the overall heap usage. To assess the effect of ctgc separately, we omitted traditional garbage collection here.

In Figure 7.2 we have the input size n on the x–axis: the lists sorted were $[n-1,\ldots,0]$. These list have the largest memory consumption and can therefore be seen as worst-case. The lines in the figure show the amount of heap used by the program without ctgc (dashed line), at the end of the ctgc version (solid line), and the maximal heap usage during execution of the ctgc version (dotted line), respectively.

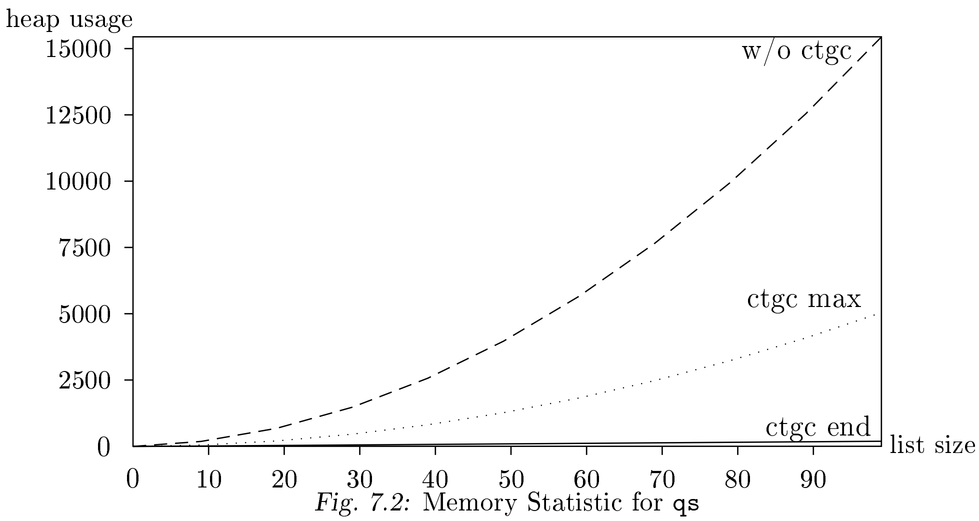


Fig. 7.2: Memory Statistic for `qs`

Note that the maximal heap usage of the ctgc version is only a third of the memory consumption of the version without optimisation. The ctgc version of the program is optimal in the sense that all intermediate data is deallocated; only the result is still in memory at the end. If `qs` would be called from a position where the input could not be accessed any more, the ctgc version would effectively be an in-place version.

The next experiment was to compare the evolution of the different heap usages during the run of the programs for a fixed input size. Here we compared the behaviour for three different list styles:

Reverse list: In Figure 7.3(a) we compared the different memory consumptions during the evaluation of `qs([9,8,...,0])`. The diagram shows the number of heap cells used at each function call performed. After an initial phase, where only alternating calls to `filter` and `qs` are executed until an empty list is reached, the two curves start to differ. The ctgc version continuously allocates and deallocates intermediate cells. The point of maximal memory consumption of the ctgc version is the end of the initial phase. Until then no deallocation is performed.

Sorted list: If we use already sorted lists as input for `qs` (see Figure 7.3(b)), the two variants differ before the point of maximal heap usage of the ctgc version is reached. For this type of list, the ctgc version uses the largest amount of heap.

Mean list: For the list $[5, 2, 1, 0, 4, 3, 8, 7, 6, 9]$ which has the property that the pivot element of each recursive call to `qs` is the mean of the remaining list, we obtain the memory behaviour shown in Figure 7.3(c). The list were generated by the function `meanmake`.

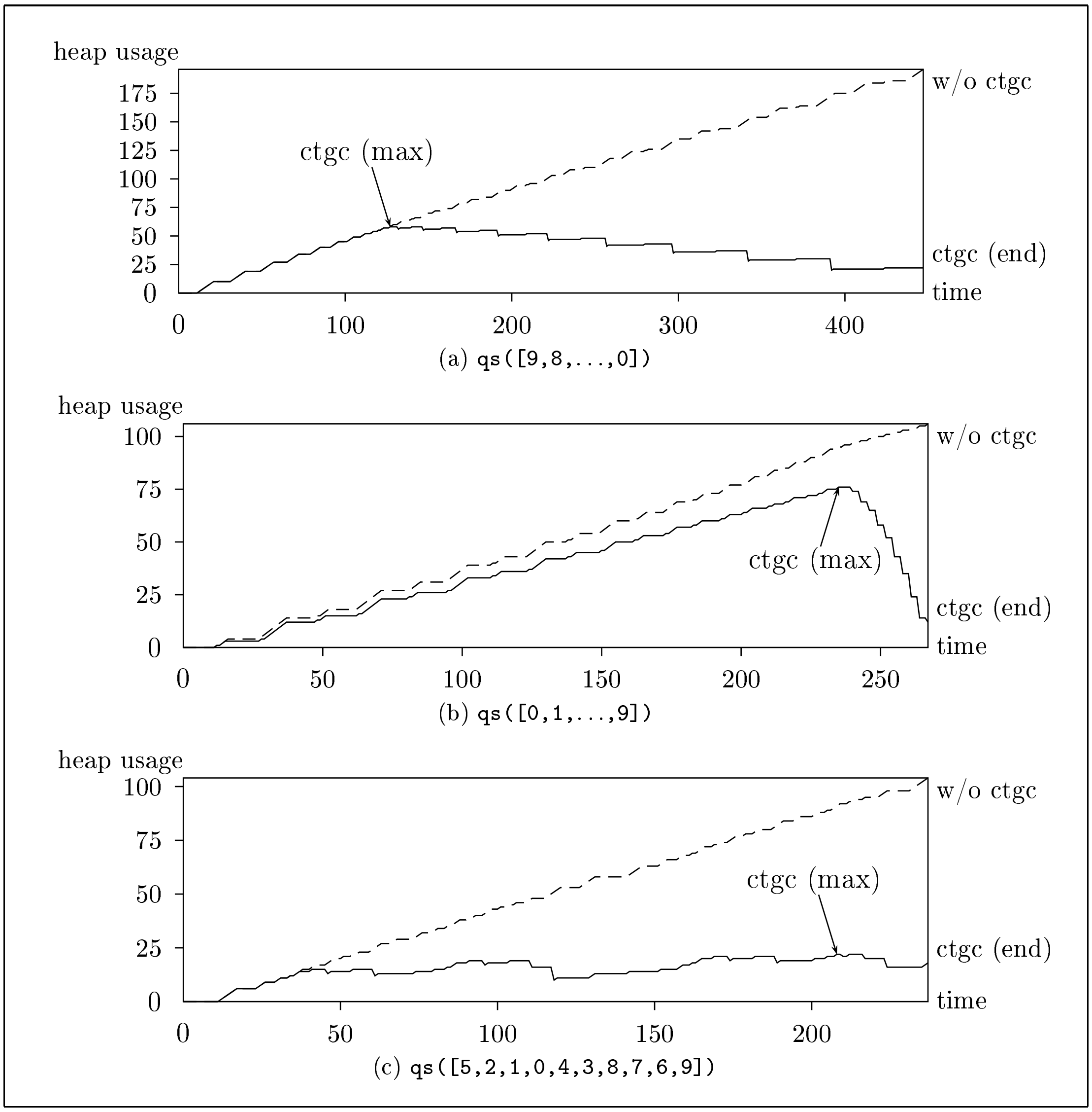
```
mm :: int -> int -> [int]
mm n o | (o<=0) = []
mm n o = (n/2+o:(mm n/2 o)++(mm (n/2)-1+(n%2) o+n/2+1))
meanmake :: int -> [int]
meanmake n = mm n 0
```

These experiments show that ctgc dramatically improves memory utilisation, even in the worst case of an already sorted list.

We now focus on the behaviour of ctgc in the presence of a garbage collector. Therefore, we used the public domain garbage collecting storage allocator that is intended to be used as a plug-in replacement for C's `malloc` and `free` [BW88, Boe91]. By simply adding this library, we get a garbage collecting C runtime system, and hence can turn any C program into a garbage collected C program. In our case, we only had to modify the file `intlist.c` which implements the constructors `NIL` and `CONS`. The library can be obtained from the URL ftp://ftp.parc.xerox.com/pub/gc/gc4.10.tar.gz.

As stated in [JL96, Chapter 9] "languages like C present a considerable challenge to the garbage collection implementer" since neither compilers nor operating systems support the collection of garbage in the runtime heap of C. The collector we have chosen is fully conservative non–moving collector based on mark and deferred sweep. It interrupts the normal execution of the program only for a limited amount of time, at the price of not reclaiming all possible garbage. Due to this choice, the results of the experiments can not directly be transfered to a full implementation of a functional language, where the garbage collector is supported by the compiler. On the other hand, it gives a first impression of the usefulness of ctgc.

In addition to the normal mode of operation, the collector also support "generational mode". In the "traditional mode", the collector interrupts client code for the duration of a garbage collection mark phase. Since this may be unacceptable if interactive response is needed for programs with large heaps, the collector can also run in a "generational" mode, in which it



(a) `qs([9,8,...,0])`

(b) `qs([0,1,...,9])`

(c) `qs([5,2,1,0,4,3,8,7,6,9])`

Fig. 7.3: Memory Behaviour of `qs` For Different List Styles With and W/O CTGC

usually attempts to collect only objects allocated since the last garbage collection. Furthermore, in this mode, garbage collections run mostly incrementally, with a small amount of work performed in response to each of a large number of `GC_malloc` requests.

Our first experiment with the combination of gc and ctgc was to compare the evolution of the memory usage of `qs` for a list of 3000 elements with gc and with gc+ctgc. This time, we included the generation of the list. The results are in Figure 7.4. The heap size is given in bytes this time. The spikes in the behaviour of the gc–only version indicate positions where garbage collection occurs. Because the garbage collector is designed to interrupt computation only for a limited amount of time, not all garbage is found during these cycles. In total, 18 collection cycles occurred during the runtime of the gc–only version, whereas the ctgc+gc version has only one. Here it becomes obvious that the garbage collector we used does not reclaim all possible memory during one collection cycle: Would that be the case, then the dashed line would drop below the solid line.

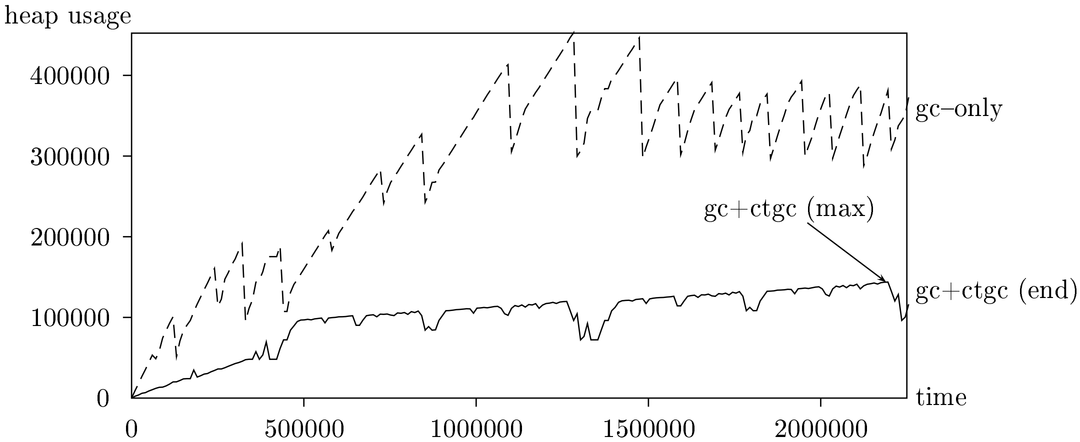


Fig. 7.4: Memory Behaviour `qs(meanmake(3000))` with gc and with gc+ctgc

The next experiment was to measure the runtime penalty imposed by ctgc. Obviously, if we compare the ctgc versions with programs without any garbage collection the native version has better runtimes, simply because of the additional deallocations in the ctgc version. To check whether ctgc is acceptable, we have to add traditional garbage collection (gc) to the runtime system both for the ctgc and for the native version.

To test whether the reduced gc frequency compensates for the runtime overhead caused by ctgc, we performed runtime test for gc and gc+ctgc versions. Unlike the memory behaviour, the runtime behaviour cannot be measured completely independent of the environment. Influences like caching, both on cpu and file system level, paging, and dynamic linking can neither be eliminated nor measured. In addition, the multi–user and networked environment we used has even more unpredictable influences like NFS latency, daemons, and other processes. We tried to minimise these effects by performing each runtime measure five times, removed the minimal and maximal values, and took the arithmetic mean of the remaining values. All runtimes are total runtimes in seconds, i.e. user time plus system time as provided by the operating system, and were performed on a SPARCstation 4 with 32 MB main memory, running with SunOS 5.5.1 and X11.

As before, we used the three different list styles, but now for varying list lengths instead of progressing evaluation:

Reverse list: The runtime of `qs([n,...,1])` for varying n can be found in Figure 7.5(a). In this case, the ctgc+gc version performs better. For list length 400, the gc–only version performs ten gc cycles, and the combined version only seven.

Sorted list: When we consider already sorted lists as input to `qs` (Figure 7.5(b)), we can see that the gc–only version has better runtime. The reason is simply that not enough gc

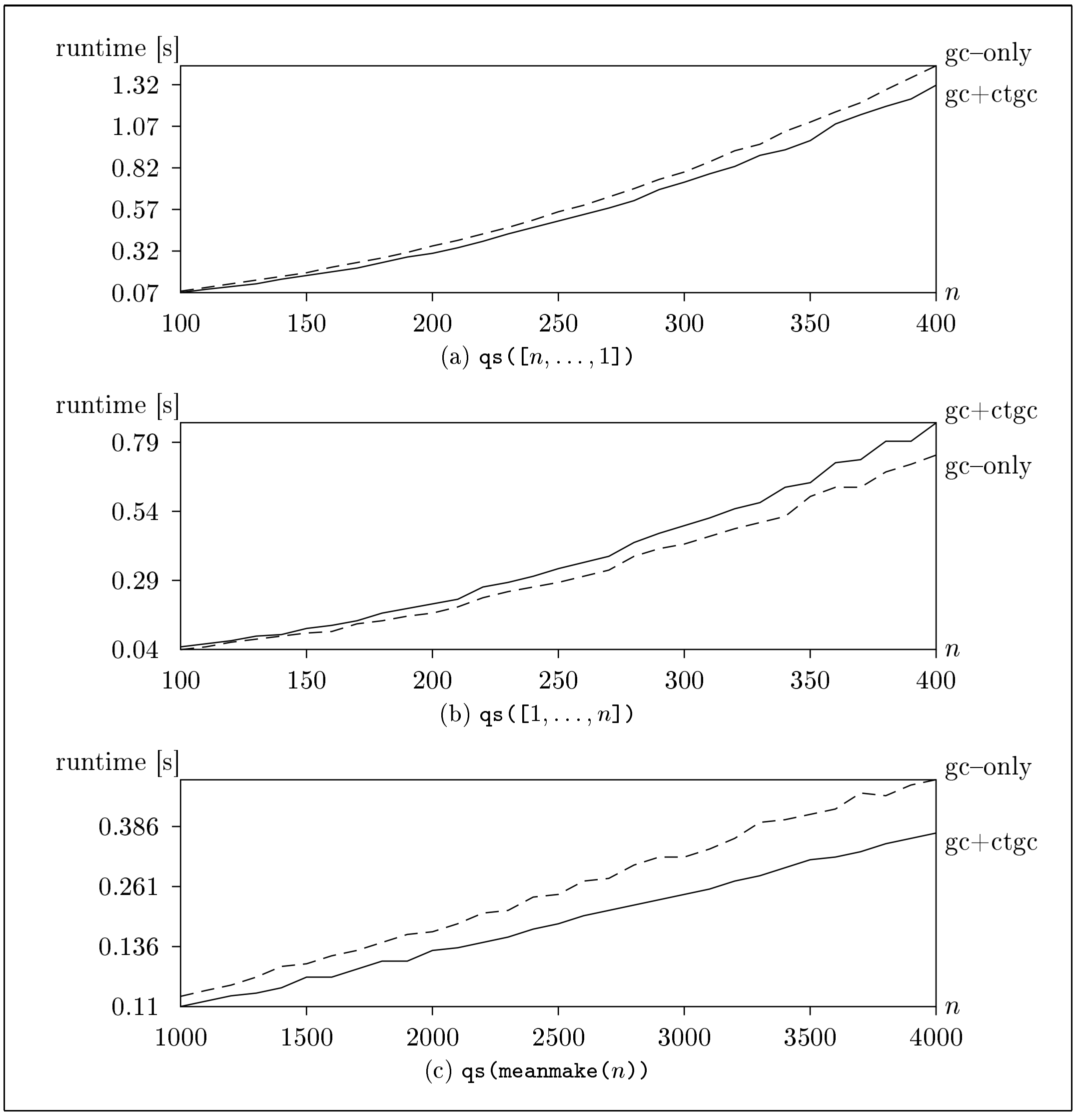


Fig. 7.5: Runtime Behaviour of `qs` GC Vs. GC+CTGC

cycles occur: The number of cycles of the combined version is at most by one smaller than the number of the gc–only version.

Mean list: Finally, for lists created by `meanmake`, we get the runtimes in Figure 7.5(c). Here number of gc cycles are: 18 for the gc–only version, and only nine for the ctgc+gc version.

In conclusion, we can see that ctgc clearly reduces the memory usage of the quicksort program. All intermediate data can be reclaimed by ctgc resulting in a space optimal version in terms of end heap usage. Although fewer garbage collection cycles were executed, the influence on the execution times was small. The overhead which is caused by ctgc compensates the time which was won by the reduced cycle frequency.

Further Benchmarks

One might argue that the quicksort program is a bad benchmark, because it is optimal for ctgc in the sense that all intermediate data can be reclaimed by ctgc. Therefore, we performed further benchmarks with various programs taken form the "NoFib Haskell benchmark suite" [Par92]. We summarised the results in Figure 7.6. For each program and input size, the four bars are the ratios between the combined and gc–only versions: maximal heap usage, end heap usage, number of garbage collection cycles, and runtime. Hence, values above 1 are "bad", and values below are "good". The programs we used for these tests are the following:

`queens` *12:* counts the number of solutions to the "12 queens" problem.

This benchmark is remarkable because it contains only a single source code location where a deallocation annotation can be inserted. Still we can see that ctgc influences memory behaviour and runtime.

`banner` *10000:* creates large version of a text with 10000 characters.

Here we have a typical situation for functional programs: A pipeline of function calls. Each of the stages of the pipeline creates a new data structure. Consequently, the result of the previous stage becomes garbage, and we can detect this situation by escape analysis. Five deallocation annotations were inserted.

`clausify` *7:* reduces a list of seven propositions to clausal form. Escape analysis found eight positions for ctgc.

`life` *1000:* computes 1000 generations of Conway's game of life. The annotated program had seven ctgc positions.

`compress` *6MB:* compresses a six MB file (the ghc-2.01 sparc executable) using the algorithm defined in [Wel84].

For the results in Figure 7.6(a) we used the traditional mode of the gc library. In Figure 7.6(b) we can find the results of the same experiments, but this time by using the incremental mode. Except for three results, the performance of the ctgc+gc version is better than that of the native version. All exceptions occurred with incremental garbage collection, and even there the loss of performance was below 15%.

The results clearly answer the question posed in [JW90], namely whether ctgc is worth the effort, in a positive way.

7.2 Avoiding Heap Allocation

The ctgc–approach introduced in the last section is truly higher–order: Heap cells created either by constructors or closures are collected. However, often we can avoid allocating the cells in the heap at all. This is possible if the point of creation of the cell is in the same incarnation of an expression as the ctgc annotation which cause the deletion of the cell. For

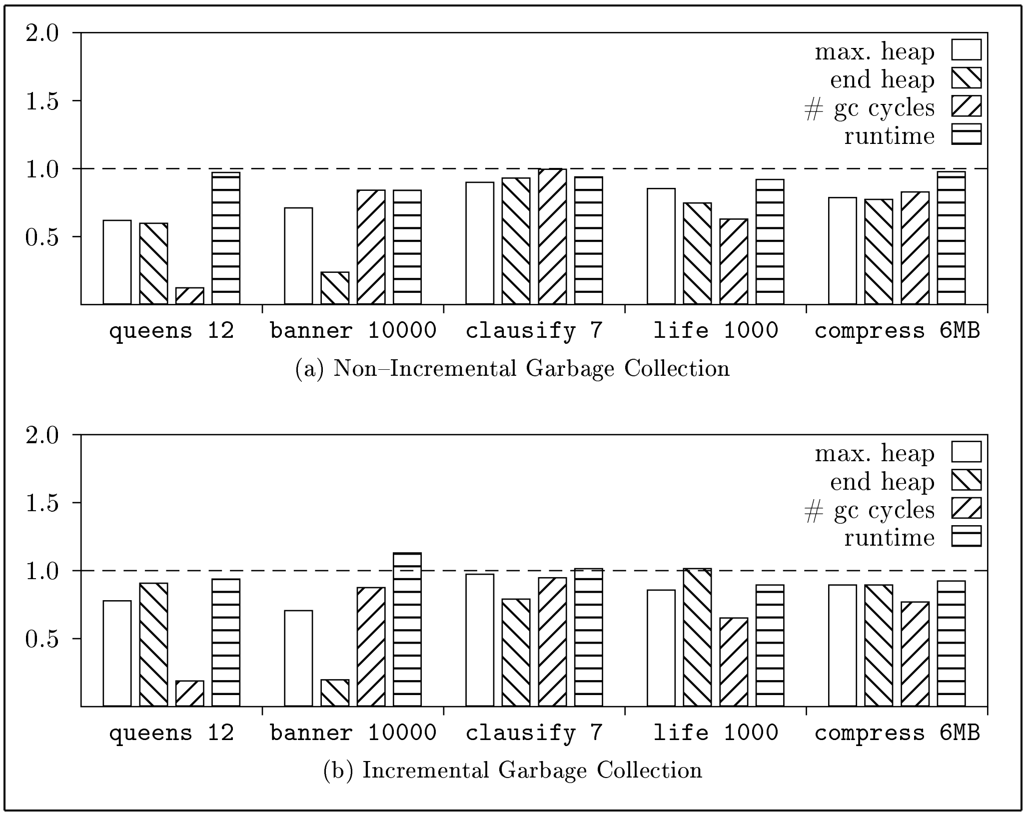


Fig. 7.6: Further Benchmarks: Ratios ctgc+gc/gc Versions

the `qs` program, we have four such situations: the partial applications `<a` and `>=a`, and the list constructors in `[a]`. In contrast, all other heap cells are created during the evaluation of recursive calls to `qs`, or calls to `++` and `filter`. Here the point of creation is in recursive calls of `qs` and in calls of `++` and `filter`, respectively.

Such cells are allocated and deallocated during the evaluation of a single incarnation of a function. Therefore, we can avoid allocating the cells on the heap and use an incarnation local environment instead.

The extensions to the graph reduction semantics which are necessary to describe these optimisations are of technical nature only, and hence we only give an informal overview of the modifications:

- Introduction of a new family of *(stack) locations* $\mathsf{SL} = \langle \mathsf{SL}^t \mid t \in \mathsf{T}(S) \rangle$ for addressing the locally allocated objects.
- Generalisation of the heap nodes HN to general nodes N, which allow not only heap locations in the nodes, but general locations $\mathsf{L} = \mathsf{HL} \cup \mathsf{SL}$.
- Modification of the notion of a heap function: Now it has to be a mapping $\mathsf{HL} \dashrightarrow \mathsf{N}$ to allow both references to heap and stack nodes.

- Extension of the graph expression semantics to use the following intermediate step:

$$\mathfrak{G}'[\![e]\!](\xi,\theta) := (l,G) \text{ where } (l,G,f) = \mathfrak{G}''[\![e]\!](\xi,\theta,[])$$

  The new semantics $\mathfrak{G}''$ has a new parameter and result component of type $\mathsf{SL} \dashrightarrow \mathsf{L}$ which stores locally allocated objects. $\mathfrak{G}''$ is defined analogously to $\mathfrak{G}$, but passes the new component as an unique component; In the same way it passes the graph component. As soon as the evaluation of an expression is finished, the definition of $\mathfrak{G}'$ ensures that the resulting local component f is discarded.

In a stack–based implementation of graph reduction, such cells can be allocated in the stack frame of the incarnation of `qs`. The above modifications model such a local frame on the level of the graph expression semantics. As soon as the evaluation of the incarnation terminates, the locally allocated objects are removed automatically.

The use of the additional component is done by a new program element, expressions of the form `local`(c) for constructors c and `local`$((e\ e_1\ \ldots\ e_n))$ for partial applications. Informally, the semantics is allocation in the local component of $\mathfrak{G}''$ instead of the heap.

Example: For the quicksort program from the last section, we obtain the following annotated version, which is equivalent to the original one:

```
    (qs (filter local((<a)) l))
 ++ ((local(Cons) a local(Nil))++(qs (filter local((>=a)) l)))
```

◇

Like ctgc–annotations, we can be sure that the annotations are semantics–preserving if the annotated object does not escape from the enclosing expression. Hence, we can use escape analysis to find appropriate positions to place the annotations. We omit the formalisations, since it is essentially the same as for the ctgc annotations.

In this particular example, we have `local()`–annotations on both constructors and partial applications. Typically, this annotations can be done for partial applications more often than for constructors, because partial applications are mainly used as parameters for higher–order functions, like `map`, `filter`, and `foldr`. There they act as parameters which determine the behaviour of such generalised schemes, but are not directly part of the result.

However, this is not possible in general. Although uncommon, we can use partial applications to simulate data structures. An example for this behaviour can be found in the `cse`–program of the NoFib benchmark suite, which performs common subexpression elimination. There we have a functional `+=>` defined as

```
+=> :: Eq a => a -> b -> (a -> b) -> a -> b
+=> x fx f y = if x==y then fx else f y
```

This function implements the update of a function `f` at a single point `x` with a new value `fx`. In the partial expression `(+=> n (head ms) r)` it is used to update a renaming `r` of labels. Here, the functional parameter `r` escapes from the partial application. The result is a closure of type `a->b`. If the result is later again parameter for `=+>`, we get a linked list of closures, like in Figure 7.7.

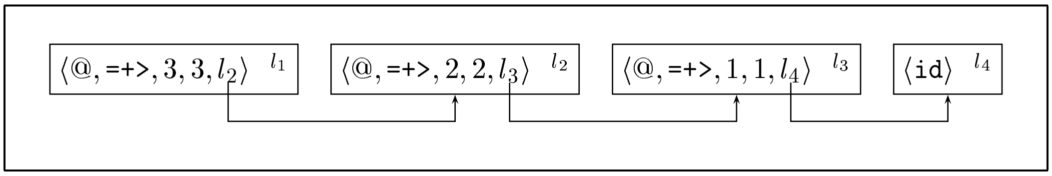


Fig. 7.7: Linked Closures created by `=+>`

7.2.1 Further Improvement: Analysing the Call Structure

By examining the call structure of the program, we can often perform a further optimisation. In the above example, the execution of `filter` will never cause another call to `qs`. Therefore, there will always be only a single *active incarnation* for `<a` resp. `>=a` in the course of the evaluation of `qs`. However, the stack allocated closures remain on the stack until termination of the enclosing `qs` call. Consequently, each recursive call of `qs` will creates new closures in its stack frame, although the already existing closures will never be used again.

Instead of allocating closures subsequently in the stack frames for the `qs` calls, we can *statically* create only one heap cell, which represents all closures for all partial applications in `qs`. By updating the locations in this heap cell, we can reuse it every time a corresponding closure is needed.

The call structure of a program can easily be analysed by examining the program text. All functions on the right hand side of a function definition are considered to be callable in one step. Similarly, all functions which have the type of a functional parameter are considered to be callable. Finally, we have to construct the transitive and reflexive closure of this one–step relation to obtain the call–structure of a program.

7.3 Summary

In this chapter we have demonstrated two applications using the knowledge of the escape behaviour: (1) compile–time garbage collection removes intermediate structures (either data or closure chains), and (2) efficient closure utilisation avoids the allocation of heap closures. Following our model–based approach, the proof of correctness depended on the semantics property escaping only, and not on the way this property is approximated. Therefore, these proofs remain valid for other methods than the escape analysis $\mathfrak{E}$.

For both applications we have introduced program annotations which allowed us to express situations where we can exploit non–escaping. We defined the graph semantics of annotated programs and showed how to obtain annotations based on the information provided by $\mathfrak{E}$.

Experimental results have shown that annotated programs have much better memory behaviour, both in terms of overall memory usage and peak usage. In combination with traditional garbage collection, also the runtimes of the programs decrease in most cases.

Part III: Extensions

8. Extensions of F

The language F we used in the preceding chapters lacks many of the advanced features of modern functional languages. In this chapter we discuss extensions of F, which allow more comfortable programming. A major focus is their effect on the abstract interpretation $\mathfrak{E}$.

8.1 Syntactic Sugar: Pattern–Matching, Local Definitions, . . .

Realistic functional languages contain syntactic sugar like pattern matching, list comprehensions, local definition of functions, or λ–abstractions. All these constructs can be removed by well–known transformations resulting in programs very similar to those expressible in F. This approach is commonly used for functional languages.

However, there is one minor difference between the core languages which are typically used in this context. While these languages have a `case`–construct which allows the simulation of pattern–matching, we have used constructor tests and selectors. Obviously, every `case`–construct can be translated into a cascading sequence of selector tests embedded in **`if`** . **`then`** . **`else`** ., where the bound variables are replaced by selector sequences. The reason why we have chosen to use the test/selector approach is that these operations are more primitive. Therefore, the abstract semantics is simpler and clearer.

8.2 Constructors with Functional Arguments

The constructors were restricted to have parameters of type $s \in S$, i.e. either basics sort or constructed sorts. This was necessary to avoid situations like `data C = K (C -> int)`. Here the constructor `K` has type $(\mathtt{C} \rightarrow \mathtt{int}) \rightarrow \mathtt{C}$. This introduces another level of recursiveness to the definition of domains for the types. For the denotational domains, this means that we have to use reflexive domains [Sto77], i.e. solutions of domain equations like $A \simeq A \rightarrow A$.

For the abstract domains, the additional recursiveness is fatal since the abstract domains can become infinite with the approach we used so far. For the above example, we would obtain $\mathsf{A}^{\mathsf{C}} = \mathbb{B} \times \mathsf{A}^{\mathsf{C} \rightarrow \mathtt{int}} = \mathbb{B} \times \mathbb{B} \times [\mathsf{A}^{\mathsf{C}} \rightarrow \mathsf{A}^{\mathtt{int}}] = \ldots$, which is not well–defined.

For the graph semantics this is no problem at all, since the constructor `K` is simply represented by a heap node containing the location of a heap node for the functional argument.

Of course, we can find intermediate levels between this extremal example and the total ban of functional constructors. One possibility is to use a *stratified approach*, where types occurring in functional arguments are required to have a lower level in the dependence hierarchy of constructed sort. This allows the use of the same mechanism as before to define denotational and abstract domains.

However, there are programs which actually require the worst case. For instance, the following type[1] can be used to represent ordinal numbers:

```
Ord = Zero | Succ Ord | Limit (int -> Ord)
```

The limit of the countable sequence $f(0), f(1), \ldots, f(n), \ldots$ can be represented as `Limit` f.

Therefore, we have to find a way to cut off this additional recursion explicitly. The idea is a direct generalisation of the general principle of the abstract domains, namely that the access to a recursive component of an abstract value a is the identity. Here, the situation is more complicated, because we have to reconstruct a functional value from a.

Given a constructor c with functional argument t, we remove all subtypes which are recursive in the way described above. Note that the resulting argument type t' may be non–functional. If this is the case, then we remove this argument completely. We then use the same mechanisms to construct the abstract domains as in Section 4.1. For our two examples, this results in the domains $\mathsf{A}^{\mathtt{C}} = \mathbb{B} \times \mathsf{A}^{\mathtt{int}} = \mathbb{B}^2$ and $\mathsf{A}^{\mathtt{Ord}} = \mathbb{B}$. In addition, the appropriate abstract constructor and selector functions become more complicated:

- For the constructor, we have to convert a functional argument in the appropriate way and ensure that the resulting abstract value reflects the worst case:

$$\begin{array}{rcl} \mathfrak{C}[\![\mathtt{K}]\!]((a,f)) & = & (a,0) \sqcup^{\mathtt{C}} \{(0, f(b)) \mid b \in \mathsf{A}^{\mathtt{C}}\} \\ \mathfrak{C}[\![\mathtt{limit}]\!]((a,f)) & = & a \sqcup^{\mathtt{Ord}} \{f(b) \mid b \in \mathsf{A}^{\mathtt{int}}\} \end{array}$$

- For selectors, given an abstract value a, we use this value as result of the function wherever the recursive type was removed. For positions, where the recursive type was removed from the functions argument, we create the function by using the same result for all arguments. As escape tag of the resulting function, we use the escape tag of the value a: $\mathfrak{C}[\![\mathtt{sel}^1\text{–}\mathtt{K}]\!]((a_1, a_2)) = (a_1, \lambda(b).a_2)$ and $\mathfrak{C}[\![\mathtt{sel}^1\text{–}\mathtt{Limit}]\!](a) = (a, \lambda(b).a)$.

8.3 Parametric Polymorphism

Parametric polymorphism allows data definitions like `List a ::= Nil | Cons a (List a)`. In addition to the monomorphic types of Chapter 2, polymorphic types are allowed to contain type variables (`a`) and type constructors of different arities (`List`). Instances of a polymorphic type can be obtained by replacing type variables with types.

Denotationally, models for a polymorphic language are more complicated. The problem is to define the domains associated with polymorphic types. The straightforward idea to model, e.g. the type `List a` as a function which maps all instances for `a` to the set for `List a` can

[1] This example is due to Simon Thompson.

lead to a transgression to Russell's paradox [RP90]. Therefore, models for polymorphic type use *partial equivalence relations (PERs)* [Gun94] or categories [See87].

Again, for the graph reduction semantics the introduction of parametric polymorphism means no fundamental changes: The boxed representation of basic values in separate heap cells implies that constructors only contain pointers to other heap nodes. Therefore, the amount of memory needed to represent a constructor node is fixed, that is it is independent of the amount needed for the representation of the entries. This "'one size fits all" approach comes to fruition for parametric polymorphism: The same kind of constructor can contain entries of different types.

Our abstract interpretation can easily be extended to handle parametric polymorphism. The main idea is that a polymorphic abstract function cannot affect data within a polymorphic component a. Hence, the abstract interpretation can use the smallest set to represent polymorphic components. For instance, we can associate the set $\{0,1\}^2$ with the polymorphic list type and evaluate each polymorphic function once with this type.

To formalise this intuition, we can use a notion introduced in [Abr86]: Our escape analysis is *polymorphically invariant*.

Definition 8.1 (Polymorphic Invariance)
Let P be a property of (monotyped) terms $\widehat{\mathsf{CT}}^t$. P is *polymorphically invariant* iff for all monomorphic instances t_1, t_2 of a polymorphic term t holds: $t_1 \in P \iff t_2 \in P$. $\lhd$

Given a polymorphic expression, our analysis will return the same results on any two monotyped instances of that expression. Therefore a polymorphic function can be analysed by considering the simplest monotyped instance of this function.

8.4 Modules

In a modular environment, where different modules can be compiled separately, it is important that optimisations can be done across module boundaries. Otherwise, the use of the optimisations is limited to the inner of a module and hence has only a local effect. While compiling a module A, we must have access to the necessary information concerning modules $B_1, \ldots, B_n$ used by module A. However, we do not have access to the source code of these modules. An optimisation which cannot be handled in this way is *deforestation* [Wad90, MW92]: For each unfold step, the definition of the function unfolded must be known.

The technique used for this aim is to write the information concerning a module B_i to an interface file while compiling B_i. If the module A using B_i is compiled later, it can use this interface file to obtain the necessary information. Of course, this approach is only correct iff the modules are not recursive and are compiled in bottom–up order of the module dependency hierarchy. For escape analysis, this means to store (descriptions of) the abstract domains and abstract functions for data types and definitions in the module.

8.5 Lazy Evaluation

Both the denotational semantics $\mathfrak{M}$ and the augmented denotational semantics $\widehat{\mathfrak{M}}$ are non–strict, and Lemma 5.5 shows that the escape analysis $\mathfrak{E}$ is a correct approximation of escaping. Hence, we can be sure that the information obtained by $\mathfrak{E}$ are correct for a non–strict language. The problem here is the graph reduction semantics: We have chosen an eager evaluation strategy because the optimisations we have done in Chapter 7 depend on the property that heap cells are garbage *after an evaluation has terminated*. For lazy evaluation, however, there is no general way to determine a fixed source code location where we can be sure that a computation has terminated: The flow of control and the program structure do not longer coincide. Hence, we have no positions where deallocations can be triggered. To combine lazy evaluation and better memory usage, we have two possibilities:

Use other memory strategies. In Chapter 7, we have described the use of explicit deallocation as memory reuse strategy. Other possibilities are *garbage marking* and *destructive reuse*. They are discussed in more detail in Chapter 9. In both cases, the point of memory reuse is moved from the application to the inside of the function. They can coexist with lazy evaluation because the heap cells are marked resp. reused as soon as they are encountered during evaluation. However, these approaches suffer from the deficiency that they require different versions for shared and unshared arguments and therefore can cause code explosion.

Use ctgc only in conjunction with strict functions. In principle, we can use strictness analysis [Myc80, BHA86a] to identify parts of the program that are strict. In these parts, we can use the results of the escape analysis as described in the previous chapter. However, this can only be done if two conditions are met:

1. The information obtained by the strictness analysis must be compatible with the information from the escape analysis. While the data abstraction of escape analysis partitions values into levels, strictness analysis for non–flat data structures [Wad87] typically uses domains which distinguish between the head and the rest of the list. It is unclear how these informations can be combined in a good way.
2. The runtime system must actually use the strictness informations. Although much work has been done to improve strictness analysis for recursive types, there is very little work on how to exploit such information. In [FB93] *evaluation transformers* are used for strict evaluation of lists. However, they report few performance benefits, and sometimes even large costs. Also in [PP93] it is reported that the use of strictness information, except for very simple data structures not including lists, is not worth the effort and in some cases the use can even have negative effects on program execution.

8.6 Summary

In this chapter we have discussed various extensions of the language F and their influence on our results: We have demonstrated how the results for the austere language F can be used as a basis for realistic functional languages.

9. Related Work

In this chapter we describe how our work relates to other published work. We consider both techniques for the analysis of programs and program optimisations based on the results of the analyses.

9.1 *Escape Analysis*

Four other variants of escape analysis are documented in the literature and all are used in the context of compile–time garbage collection. All of them are far more restrictive with respect to data structures. The only recursive data structure which is possible with these approaches are list structures, arbitrary recursive data structures are not considered. In Chapter 4 we have seen that the addition of arbitrary recursive data structures leads to non–trivial problems. If correctness is considered at all, it is done with respect to an abstract machine.

9.1.1 *Goldberg & Park's Escape Analysis*

The escape analysis by Goldberg & Park [GP90, PG92] uses the same idea as ours to obtain the abstract domains: The abstraction consists of representing a list by the *levels* of a list. Within a level, all constructors are assumed to behave in the same way.

For a list type t with d levels, the abstract value is essentially either $(1, i)$ with $i \in \{0, 1, \ldots, d-1\}$ indicating that only the lower $i+1$ levels of the list may escape, or the least element $(0, 0)$ indicating that nothing escapes. In addition all abstract values have a component describing their abstract functional behaviour. It is *err* for non–functional types. For lists of integers, we have the following domain:

$$\begin{array}{c} ((1,1), err) \\ | \\ ((1,0), err) \\ | \\ ((0,0), err) \end{array}$$

The underlying observation is that for functional programs, if a level d escapes, then also all levels $d + k$ *below* d. Obviously, it is simply not possible to define a function, e.g. on `ListOfList` where only the top level spine and the `int` entries escape but *not* the spines of the `int`–lists in between.

However, Goldberg & Park's analysis differ in the abstraction of functional types. Although in both cases the abstraction of a function is a tuple (t, f), where f is an abstract function, the objective of the first component t is completely different:

1. Goldberg & Park use it to take free variables from the definition of the function into account. This yields more precise information on the escape behaviour of the *parameters of a partial application.*

2. We use t as a binary tag for the escape behaviour of the *closure* representing the function. Only this feature allows to perform collection of closures or stack allocation of closures as described in Section 7.2.

Correctness of the analysis is not formally validated.

Complexity

In [Deu97] the complexity of Goldberg and Park's analysis is analysed. Caused by their choice of abstract domains, this relationship is far more complex than for our abstract domains: For non–functional types t the domain is a chain and hence *all* elements are join–irreducible. Moreover, the analysis does only give additive function for first–order programs. Already second–order function are not necessarily additive in their functional arguments.

However, Deutsch shows that all first–order functions $f^\sharp$ of type $t_1, \ldots, t_n \to t$ generated by Goldberg & Park's analysis are additive in its range. Given that, he can use the results by Nielson & Nielson to reduce complexity: The number of join–irreducible elements of the range of $f^\sharp$ is less or equal $n \cdot d$, where d is the size of the largest abstract domain. This results in essentially the same reduction as for $\mathfrak{E}$. The number of function evaluations to find the fixpoint for a program with p definitions is bounded by

$$\#\text{eval} = p^2 \cdot n^2 \cdot d^3$$

Furthermore, $f^\sharp$ can be completely characterised by its values at $(\bot, \ldots, \bot)$ and $(\top, \bot, \ldots, \bot)$, $(\bot, \top, \ldots, \bot)$, $\ldots$, $(\bot, \ldots, \bot, \top)$ by a property similar to the Shannon expansion of boolean functions or affine transformations in linear algebra:

$$\begin{array}{rcll} f^\sharp(x_1, \ldots, x_n) & = & & f^\sharp(\bot, \ldots, \bot) \\ & & \sqcup & (x_1 \sqcap f^\sharp(\top, \bot, \ldots, \bot)) \\ & & \sqcup & \ldots \\ & & \sqcup & (x_n \sqcap f^\sharp(\bot, \ldots, \bot, \top)) \end{array}$$

Hence, we can reduce the number of points where $f^\sharp$ must be evaluated to $n+1$. Although this yields a reduction in the number of iterations

$$\#\text{eval} = p^2 \cdot n \cdot (n+1) \cdot d^2$$

it also causes an increase of the cost of each evaluation. For each application in each right hand side, the above expansion must be performed.

Based on the observation that the domains form a chain for the first–order case, it is possible to reduce the complexity even further:

1. Definition of a backward analysis, which is equivalent to the original one. It uses (first–order) domains where the functional component is separated.
2. Using an algorithm of complexity $O(n \log n)$, the backward analysis of a program of size n can be expressed as a system S of algebraic equations of the semi–ring $(\mathbb{N}, \max, \min)$.
3. By using an algorithm by Knuth [Knu77], the system S can be solved in $O(|S| \log |S|)$. Note that this improvement is heavily based on the abstract domains being chains. Only this property allows the use of Knuth's algorithms.

In our opinion, the difference in the worst–case complexity for the first–order case is not worth the effort resulting from these techniques.

For the second–order case the situation is far worse. Is it not possible to reduce the complexity of the naïve algorithm by using additivity. Moreover, it is shown to be DEXPTIME–hard! Hence, no method for obtaining equivalent information can have better complexity.

Relative Expressive Power

The relative expressive power of the two analyses will be discussed in more detail in [Moh]. Here, we only summarise the results.

First we consider the case, where no functional arguments or results occur. Obviously, our analysis has a greater expressive power than that of Goldberg & Park in the sense that our domains are not based on the assumption that if a level d escapes, then also all levels $d + k$ *below* d. For instance, $(1, 0) \in \mathsf{A}^{\mathtt{ListOfInt}}$, which indicates that the spine of a list escapes but not the entries, has no counterpart.

However, this difference in expressive power is not relevant for purely functional programs: Those values which are not expressible by Goldberg & Park's domains cannot be result of $\mathfrak{E}$, provided the environments do not contain such values either. Given that, it is easy to show that both analyses are equivalent.

As a consequence of complexity results we know that $\mathfrak{E}$ cannot be as precise as the analysis by Goldberg & Park: Any method which is as precise must require at least deterministic exponential time for second–order functions. Since $\mathfrak{E}$ uses quadratic time, it cannot have the same precision. Examples for this loss in precision are arguments of partial applications: $\mathfrak{E}$ cannot determine escape information for arguments of partial applications. However, the analysis by Goldberg & Park uses the first component of functional abstractions to capture information about free variables from the definition of the function.

Using the formalisation from [JM81], our analysis can be seen as *attribute independent*, while Goldberg & Park's analysis is *relational*. There it was shown that attribute independent flow analysis has better complexity than relational one at the price of less precise information.

On the other hand our analysis can infer information on the closures for functional parameters. This is not possible with the analysis by Goldberg & Park. Consequently, the analyses are incomparable for the case that higher–order functions are permitted. However, we believe that the stack–allocation of closures has more practical relevance than the removal of arguments of partial applications.

9.1.2 Hughes' Inheritance Analysis

In [Hug92] Hughes introduced an analysis (called *inheritance analysis*) to extract escape information from higher–order functional programs. It is very similar to the one by Goldberg & Park since it uses essentially the same abstract domains. Furthermore, correctness is proved wrt. an abstract machine model.

9.1.3 Jones & Le Métayer's Transmission Analysis

The abstract domains in [JM89] are introduced as infinite domains. To obtain finite domains, the height of the abstract list must be restricted. However, there is no general scheme how to choose such a threshold. Instead, Jones & Le Métayer report that "in practice ... [choosing two levels] is most often enough".

9.1.4 The Analysis by Inoue, Seki, and Yagi

A different approach is taken in [ISY88] to detect non–escaping occurrences in a functional program. The abstract interpretation is done by translating the original program to a context–free grammar. The language generated by a defined function contains the escape information for this function.

9.2 Flow Analysis

The aim of *flow analysis* (aka *closure analysis* or *control flow analysis*) is to determine for each expression e of a program a safe approximation of the set of expressions which may have originated the value of e. Hence, it can be seen as a *backward approach* to escape analysis: Starting from the complete expression flow analysis tries to find subterms while escape analysis starts from the subterms.

The first approaches [Ses91, Shi91] were based on abstract interpretation, but suffered from the deficiency that it was not possible to perform the analysis on module level. Since our escape analysis is well–suited for a modular environment, we suspect that this is mainly caused by a wrong flow of information. Recently [Fax95, Ban97], the focus has shifted to *type based* systems where the analysis is formulated as an inference problem. The advantage for flow analysis is that modular analysis becomes possible. Apparently, all type systems can be seen as abstract interpretations [Cou97].

9.3 Compile–Time Garbage Collection

Besides the different approaches to escape analysis, several other analyses for reducing the runtime memory consumption by ctgc have been studied:

- In [Hug92] a combination of *generation analysis* and escape analysis is used. The generation analysis determined which parts of the result can actually be generated by the expression and hence can avoid searching for the allocated cells.

- A combination of sharing analysis and escape analysis is used in [ISY88]. The underlying analyses use context-free grammars.
- A backward analysis which is essentially abstract reference counting is used in [JM90].
- Information on whether a particular cell is going to be accessed in the future is inferred by the necessity analysis described in [HJ90].

In [JM89] a different approach is taken: List constructors which are not shared are collected as soon as they are dereferenced, i.e. as soon as they match a constructor on the left hand side of a function definition. The abstract domains are introduced as (infinite) domains $I_{\texttt{list}}$. For practical applications, the height of the abstract list is restricted, the choice of a threshold is left to the user. Additionally, if it is detected that a deallocation is immediately followed by an allocation, update-in-place is performed.

In [Ham95] a classification of three different memory reuse strategies is introduced:

Garbage marking: Cells which will become garbage after their first use are marked at their allocation. After its use a cell is returned to the heap. This allows the use of ctgc in combination with lazy evaluation.

Destructive allocation: Without being returned to the memory manager a cell which is used and garbage is reused instead of a new allocation.

Explicit deallocation: This strategy is used in this thesis and consists of returning the cell to the heap at a certain program point.

Surprisingly few studies exist on the effect of ctgc on memory and runtime behaviour. The only work to our knowledge is [Jon95]. But there only one program, the Conway number package is studied with the method described in [JM89]. Jones concludes that ctgc is not worth the effort[1], since they did not observe the expected reduction in *runtime*. We doubt this conclusion for two reasons:

1. The usage of ctgc to reduce runtime is the wrong approach. As our experiments have shown, we may be lucky to get a reduction in runtime, but the major task is a better memory behaviour as long as the runtime is not affected in a negative way.
2. The Conway number package is a bad choice. By its nature it is a library without much intermediate data which could be avoided or reused. Therefore we doubt that the results can be transfered to realistic programs. This problem was already seen by Jones.

9.4 Uniqueness Type System

The language Clean [BvELP87, AP95] uses a type system with uniqueness information. The type system [SBvEP94, BS95] allows the distinction between unique and non–unique

[1] Maybe a self–fulfilling prophecy [JW90]?

objects. Objects which are identified to be unique essentially have the property that their graph representation is unshared during graph rewriting.

The relation to ctgc is obvious: Unshared objects can be updated destructively. In fact, the type system uses sharing analysis [AP95] to determine the uniqueness information. However, this approach avoids the problems occurring with sharing based ctgc: Since the property is build into the language, we avoid the code explosion problem.

Furthermore, the uniqueness type system allows an elegant approach to perform I/O in functional languages. The Clean I/O system is based on passing the "world" as an explicit unique parameter of functions. In contrast to the monad based I/O of Haskell [PW92] the programmer is not forced to fix the evaluation order of the complete interactive top spine of the program. Consequently, I/O in Clean disturbs the functional structure much less than I/O in monad–based languages does.

9.5 Region Inference

Region inference [TT94, Tof95, BTV96] is an analysis for inferring safe approximations of the lifetime of values created during the execution of strict functional programs. The basic idea is to use a different memory model where the store consists of a stack of *regions* instead of a single heap. Region inference determines (1) at which points regions must be allocated on the region stack, (2) at which points complete regions can be deleted, and (3) for each expression producing a value which region to put the result in. A special feature of the region inference based memory layout is that it allows dangling pointers: Regions may be deleted although there exist active pointers into that region. This is possible since the analysis can infer that the corresponding value in the region being deleted is not going to be accessed again.

Since complete regions are deleted, this approach seems to be a deletion strategy, in contrast to the retention strategy of classical heap storage. However, each region is a heap capable of holding an unbounded number of cells. Consequently, the size of a region cannot be fixed at the time it is allocated. Therefore, the region stack is not really a normal runtime stack, but it must be mapped to a normal heap. In fact, each region can have the same memory behaviour as a normal heap, including the necessity for garbage collection. The resulting memory behaviour on the heap is very similar to that of ctgc; at certain points in the program regions are deleted which result in the deletion of portions of the heap.

9.6 Deforestation

Deforestation is a technique introduced by Wadler [Wad84, Wad86, Wad90] for first–order languages. Its aim is also the removal of intermediate data structures, but in contrast to the other approaches discussed in this chapter it is a *program transformation*. Based on the unfold/fold approach by Burstall and Darlington [BD77], intermediate data structures are removed by unfolding both the producing function and the consuming function such that constructor and selector directly meet.

Although much work has been done on extending the original approach by Wadler to cope with more general cases [MW92, Chi94, Ham96] deforestation has several major disadvantages:

1. Since it is based on unfolding functions, it cannot be used in a modular environment with separate compilation.
2. To find the right fold, an implementation must record all intermediate unfoldings, which introduces substantial costs and complexity.
3. It is not easy to stop unfolding at all.

Especially the last item is the major obstacle. In fact, it has become the main topic of research on deforestation [FW88, Chi94, Sør94, Sei96, SS97].

9.6.1 Short Cut Deforestation

The drawbacks of "full" deforestation lead to the development of "short cut" deforestation [GLP93]. It assumes that functions for consuming lists are expressed by the primitive `foldr` and functions for producing lists are expressed in terms of the primitive `build`. After inlining producer and consumer the resulting `foldr`/`build` pairs can be removed. This technique was also implemented in the Glasgow Haskell Compiler.

The obvious drawback of this approach is that it requires the functions to be formulated in terms of `build` and `foldr`. Recently there have been approaches to turn recursive definitions into a suitable form [TM95, HIT96] automatically.

10. Conclusions and Future Work

In this thesis we have presented and systematically analysed a method for extracting escape information from functional programs. We have verified the correctness of this analysis and of its applications, and measured the influence on the runtime behaviour of programs. Following a model–based approach for verifying the correctness, the thesis is divided into three major parts.

In the first part, we have introduced the language F, which served as the basis of our investigations. Since escaping cannot be expressed by a standard denotational semantics of F, we have introduced a conservative extension of the standard semantics. It uses augmented domains where the standard domains are extended by binary tags. The escape semantics was defined as an abstract interpretation. The design criteria for the abstraction were based on (1) the removal of concrete values in data structures, and (2) the compression of unbounded or infinite values to finite abstract values. The compression is based on the assumption that all constructors of the same level behave in the same way. Furthermore, we have demonstrated that the design of the abstract interpretation results in quadratic complexity for the computation of the information. Finally, we have proved that the escape analysis is a safe approximation of the augmented semantics.

In the second part of the thesis, we have demonstrated the usage of escape information in program optimisation. To formalise this, we have defined a denotational model of graph reduction. The main idea was to implement functions on terms by functions on locations and graphs. Since our interest is focused on the improvement of memory behaviour, we had to model the graph semantics in a way which allowed the notion of garbage. Therefore, we had to define the graph domain as a quasi ordered set: The order on the graphs had to ignore garbage, because otherwise removal of garbage would not be a monotonic operation. Because the standard theory of denotational semantics based on the fixpoint theorem of Knaster and Tarski requires complete partially ordered sets as denotational domains, we have extended this theory accordingly. We have shown that the graph semantics is sound and that escaping is a precise model of reachability from the result in the graph semantics. The denotational approach has allowed us to formulate these proofs without having to argue about the translation of recursion to iteration. We concluded the second part of the thesis by giving two applications using the knowledge of the escape behaviour: compile–time garbage collection removes intermediate structures and efficient closure utilisation avoids the allocation of heap closures. Following our model–based approach, the proof of correctness depended only on the semantics property escaping, and not on the way this property is approximated. Therefore, these proofs remain valid for other methods than our escape analysis. For both applications we have introduced program annotations which allowed us to express situations where we can exploit non–escaping. We defined the graph semantics of annotated programs and showed how to obtain annotations based on the information provided by the abstract interpretation. Experimental results have shown that annotated programs expose much better memory behaviour, both in terms of overall memory usage and peak usage. In combination with traditional garbage collection, also the runtimes of the programs decrease in most cases.

In the third part of the thesis, we have discussed various extensions of the language F and their influence on our results: We have demonstrated how the results for the austere language F can be used as a basis for realistic functional languages. Furthermore, we have discussed how our work relates to other published work. We have considered both techniques for the analysis of programs and program optimisations based on the results of the analyses.

From a theoretical perspective, the main contributions of this thesis are:

- The notion of escaping as a property of programs based on an augmented denotational semantics.
- The denotational model of graph reduction, based on an extension of the fixpoint theorem of Knaster and Tarski.
- The first model–based proof of correctness of escape analysis and applications based on escape information.

From a practical perspective, we mention:

- The low compile–time overhead imposed by the abstract interpretation, which can be computed in quadratic time.
- The performance evaluations, which do not only demonstrate the usefulness of escape analysis, but are also the most detailed which were documented so far.
- The extensibility of the applications to realistic languages, especially to use them in modular environments.

The following sections describe some interesting topics that certainly deserve further investigation, outlining some preliminary ideas in each case.

10.1 Lazy Semantics on Quasi Ordered Sets

To model escaping as semantic property, we used augmented domains, which were obtained by adding additional binary tags to the standard domains. At least, they were *conceptually* obtained this way. In fact, we had to define the new domains explicitly without being able to reuse the standard domains.

However, we could model augmented domains as quasi ordered sets by adding the annotations as additional components to the original domains. In [Moh97], we used this technique to model the problem of dead code elimination in the simple imperative language `while`. However, the notions introduced in Section 6.4 are not suitable to cater for lazy semantics.

This is caused by the notion of convergence (Definition 6.16): We require that the values become equal at some point. Obviously, a lazy semantics cannot guarantee that and therefore the sequence of successive applications is not convergent for a lazy transformation.

Therefore, it would be convenient to generalise the fixpoint theorem even further. An approach would be to use a more general notion of convergence. Until now, we have not restricted the kind of qos we use as domains or ranges of the functions. If we assume that the range is a metric space (M, ϱ) itself we could define a finer notion: A sequence of functions $(f_n)_{n \in \mathbb{N}}$, $f_n : A \to M$ is called *convergent to* $f : A \to M$ ($\lim\limits_{n \to \infty} f_n = f$) iff for all $x \in A$ and $\varepsilon > 0$ exists $i \in \mathbb{N}$ such that $\varrho(f(x), f_{i+j}(x)) < \varepsilon$ for all $j \in \mathbb{N}$.

Unfortunately, this does not directly lead to a solution for our problem: In our case, the metric has to measure the difference in definiteness of augmented terms (a', t'). Obviously, the augmentation would not influence this difference. Therefore it would not fulfill the condition $\varrho((a,t),(a',t')) = 0$ iff $(a,t) = (a',t')$ but only $\varrho((a,t),(a',t')) = 0$ iff $t = t'$ and hence not be a metric.

However, we are optimistic that we can overcome this obstacle and obtain a generalisation of the fixpoint theorem along these lines.

10.2 Cost Semantics Based on Denotational Graph Semantics

For our applications, it was obvious that the costs in terms of heap usage were not increased. For other optimisations, this may not be the case. Therefore, it would be interesting to have a notion which would allow to prove that these optimisation indeed improve the program's execution costs. For this purpose we could use the graph semantics, since it allows a direct measure of heap cells allocated.

In [San95] a natural operational semantics for the lazy λ–calculus is extended with the notion of *cost*. However, it suffers from some deficiencies, which are related to the task of proving that applicative bisimulation is equivalent to observational equivalence.

10.3 Compile–Time Garbage Collection for Lazy Functional Languages

We have seen that the combination of ctgc and lazy evaluation is not a trivial task. The major problem is that in general it is not possible to associate a source code location with the event that the evaluation of a function is terminated. This location is needed to place deallocation instructions there. However, it might be possible to avoid the need for a source code location by passing the deallocation instructions as a "continuation–like" parameter to the functions. This additional parameter would be modelled as a function transforming the graph and would be executed after termination of the function.

For example, consider the expression $e =$ `(append [x] l)`. Obviously, we can deallocate the constructors of the list `[x]` after termination of `append`. However, if this expression is embedded in a larger one, `append` terminates as soon as e is evaluated fully, which may be caused by an expression textually far away. In addition, if e is not fully evaluated then the constructors of the list `[x]` can also be deallocated if the closure for `append` is collected by a garbage collection cycle.

We could model `append` by the following graph function:

$$app : \mathsf{P}^{\texttt{ListOfInt}} \times \mathsf{P}^{\texttt{ListOfInt}} \times [\mathsf{G} \to \mathsf{G}] \to \mathsf{P}^{\texttt{ListOfInt}} \times \mathsf{G}$$

$$app(l, l', t, G)) = \begin{cases} (l', t(g_f, L_f)) & \text{if } (g_f, L_f) = \mathsf{force}^{\texttt{ListOfInt}}(l, G), \\ & \quad g_f(l) = \langle \texttt{Nil} \rangle \\ (l_r, (g_r, L_r)) & \text{if } (g_f, L_f) = \mathsf{force}^{\texttt{ListOfInt}}(l, G), \\ & \quad g_f(l) = \langle \texttt{Cons}, l_a, l_t \rangle \\ & \quad g_r = g_f[l_d / \langle !, \texttt{append}, l_t, l', T \rangle, l_r / \langle \texttt{Cons}, l_a, l_d \rangle], \\ & \quad L_r = L \cup \{l_d, l_f\} \\ (\mathsf{NULL}^{\texttt{ListOfInt}}, (g_\emptyset, \emptyset)) & \text{otherwise} \end{cases}$$

Here, we assume that we have thunks $\langle !, . \rangle$ as fifth kind of heap nodes and $\mathsf{force}^t : \mathsf{P}^t \times \mathsf{G} \to \mathsf{G}$ forces the evaluation of thunks to head normal form. The additional parameter t is the finalisation code transforming the final graph. It the above example, it would remove the constructors for `[x]`.

Although this is straightforward for this simple example it is not yet clear how this can be done for functions on non–linear data structures like trees.

10.4 *Compile–Time Garbage Collection for* Java

Two of the major design criteria of the imperative language Java [Fla97] were robustness and security. Therefore, the designers of Java abandoned pointers and programmer–controlled memory management: A Java program can create objects via `new`, but explicit deallocation of objects is not possible. Instead, it features an automatic memory management using garbage collection.

Like in functional languages we can identify objects which cannot escape from a certain context; in this case, the context is a surrounding block of statements. For instance, consider the following (rather artificial) example of a Java class definition[1].

```
class C { public int m(void) { Object o = new Object(); return 1; } }
```

Obviously, the object created by `new` can neither leave the method `m` nor be referenced after termination of `m`. Hence, it will always be garbage. If we can define an escape analysis for Java this situation can be detected.

The main problem is the exploitation of this information. Following the security constraints imposed on the language, also the abstract machine executing compiled programs [LY97] has no possibility to deallocate objects explicitly. Hence, to use escape information for Java programs, the Java Virtual Machine must be extended to allow the deallocation of objects when they are garbage.

[1] In Java, every variable (aka field) or procedure (aka method) must be member of a class. Neither stand alone procedures nor global variables are permitted.

Appendix

A. Universal Algebra

This appendix contains some basic notions from universal algebra.

Definition A.1 (Family)
Let I be a set of indices, and A_i be sets for $i \in I$. The *family* $\langle A^i \mid i \in I \rangle$ is a mapping $\varphi : I \to \{A_i \mid i \in I\}$ such that $\varphi(i) = A_i$. $\triangleleft$

Definition A.2 (Partially Ordered Set)
Let A be a set and $\leq \subseteq A \times A$ be a relation on A. The structure $\langle A, \leq \rangle$ is called *partially ordered set (pos)* iff

1. $\leq$ is reflexive: $a \leq a$ for all $a \in A$.
2. $\leq$ is transitive: if $a \leq b$ and $b \leq c$ then also $a \leq c$ for all $a, b, c \in A$.
3. $\leq$ is anti–symmetric: $a \leq b$ and $b \leq a$ iff $a = b$ for all $a, b \in A$. $\triangleleft$

Lemma A.1 (Product POS, Function Space POS)
Let $\langle A_1, \leq_1 \rangle$ and $\langle A_2, \leq_2 \rangle$ be pos.

- The *product pos* $\langle A_1, \leq_1 \rangle \times \langle A_2, \leq_2 \rangle := \langle A_1 \times A_2, \leq_{1\times 2} \rangle$, where $\leq_{1\times 2}$ is defined as $(a_1, a_2) \leq_{1\times 2} (a'_1, a'_2)$ iff $a_i \leq_i a'_i$ $(1 \leq i \leq 2)$, is a pos.
- The *function space pos* $[\langle A_1, \leq_1 \rangle \to \langle A_2, \leq_2 \rangle] := \langle [A_1 \to A_2], \leq_{\to 2} \rangle$, where $\leq_{\to 2}$ is defined as $f \leq_{\to 2} g$ iff $f(a_1) \leq_1 g(a_1)$ for all $a_1 \in A_1$, is a pos. $\square$

Definition A.3 (Directed Set)
Let $\langle A, \leq \rangle$ be a partially ordered set and $T \subseteq A$ a set. T is called *directed* iff $T \neq \emptyset$ and for all $a, b \in T$ exists $c \in T$ such that $a \leq c$ and $b \leq c$. $\triangleleft$

Definition A.4 (Least Upper Bound)
Let $\langle A, \leq \rangle$ be a pos and $T \subseteq A$ be a set. An element $a \in A$ is called *least upper bound of* T iff $a' \leq a$ for all $a' \in T$ and for all $a'' \in A$ with $a' \leq a''$ for all $a' \in T$ holds that $a \leq a''$. We write $a = \bigsqcup T$, if it exists. $\triangleleft$

Definition A.5 (Complete Partially Ordered Set)
A pos $\langle A, \leq \rangle$ is called *complete partially ordered set (cpo)* iff

1. $\langle A, \leq \rangle$ has a least element $\bot \in A$ with $\bot \leq a$ for all $a \in A$.
2. $\bigsqcup T$ exists for all directed sets $T \subseteq A$. $\lhd$

Definition A.6 (Monotonic Function, Continuous Function)
Let $\langle A_1, \leq_1 \rangle$ and $\langle A_2, \leq_2 \rangle$ be cpos. A function $f : A_1 \to A_2$ is called

monotonic iff for all $x, y \in A$ with $x \leq_1 y$ holds that $f(x) \leq_2 f(y)$.

continuous iff f is monotonic and for all directed sets $T \subseteq A$ holds that $f(\bigsqcup T) = \bigsqcup f(T)$. $\lhd$

Theorem A.1 (Fixpoint Theorem of Knaster and Tarski)
If $\langle A, \leq \rangle$ is a cpo and $f : A \to A$ a continuous function then f has a least fixpoint $\mathsf{fix}(f) \in A$:

$$\mathsf{fix}(f) = \bigsqcup \{f^i(\bot) \mid i \in \mathbb{N}\} \qquad \square$$

Proof Given that f is monotonic, we know that $\{f^i(\bot) \mid i \in \mathbb{N}\}$ is a directed set and hence $\bigsqcup \{f^i(\bot) \mid i \in \mathbb{N}\}$ exists.
Furthermore, it is a fixpoint of f, because f is continuous:

$$\begin{aligned} f(\bigsqcup \{f^i(\bot) \mid i \in \mathbb{N}\}) &= \bigsqcup \{f^{i+1}(\bot) \mid i \in \mathbb{N}\}) \\ &= \bigsqcup \{f^i(\bot) \mid i \in \mathbb{N}\}) \end{aligned}$$

Let a be another fixpoint of f: $f(a) = a$. Because $\bot \leq a$ and f is monotonic, we have $f^i(\bot) \leq f^i(a) = a$ for all $i \in \mathbb{N}$ and hence $\bigsqcup \{f^i(\bot) \mid i \in \mathbb{N}\}) \leq a$. q.e.d.

B. Symbols and Notations

F: source language, *6*

BS: set of basic sorts, *6*

CS: set of constructed sorts, *6*

S: all sorts: $S = BS \uplus CS$, *6*

$\mathsf{T}(S)$: types over S, *6*

X: family of variables, *6*

DF: family of defined functions, *6*

BF: family of basic functions, *6*

C: family of constructors, *6*

C^{cs}: Constructors of sort cs, *7*

$CTest$: family of constructor tests, *7*

$CSel$: family of selectors, *7*

Ω: family of intrinsic functions, *7*

E: family of expressions, *8*

V^{bs}: sets for basic sorts, *8*

$T{\downarrow}$: down closure of T, *8*

$\mathsf{Id}(\langle A, \leq\rangle)$: cpo of ideals over po $\langle A, \leq\rangle$, *9*

CT^t: semantic domains, *9*

$\leq^t$: order on CT^t, *9*

$\perp^t$: least element of CT^t, *9*

PT^{cs}: partial terms, *9*

$\leq_\perp^{cs}$: order on PT^{cs}, *9*

$\perp^{cs}$: least element of PT^{cs}, *9*

CT^{cs}: infinite terms, *9*

$\mathsf{Env}(A, B)$: set of environments over A and B, *11*

BOps: family of basic operations, *11*

Ops: family of basic operations, *11*

ϱ: interpretation of basic functions, *11*

$\mathfrak{M}[\![f]\!]$: semantics of intrinsic function f, *11*

β: environment for variables, *12*

σ: environment for defined functions, *12*

$\mathfrak{M}[\![e]\!](\beta, \sigma)$: semantics of expression e, *12*

$\mathfrak{M}[\![P]\!]$: semantics of program P, *13*

$\Phi_{\mathfrak{M},P}$: semantic transformation for program P, *13*

$\widehat{\mathsf{CT}}^t$: augmented semantic domains, *15*

$\widehat{\leq}^t$: order on $\widehat{\mathsf{CT}}^t$, *15*

$\widehat{\perp}^t$: least element of $\widehat{\mathsf{CT}}^t$, *15*

$\widehat{\mathsf{PT}}^{cs}$: augmented partial terms, *16*

$\widehat{\leq}_\perp^{cs}$: order on $\widehat{\mathsf{PT}}^{cs}$, *16*

$\perp^{cs}$: least element of $\widehat{\mathsf{PT}}^{cs}$, *16*

$\oplus$: add (void) augmentation, *19*

$\ominus$: remove augmentation, *19*

$\oplus_\perp$: add (void) augmentation for $\widehat{\mathsf{PT}}^{cs}$, *19*

$\ominus_\perp$: remove augmentation for $\widehat{\mathsf{PT}}^{cs}$, *19*

$\widehat{\mathfrak{M}}[\![f]\!]$: semantics of intrinsic function f, *22*

$\widehat{\mathsf{Ops}}$: family of annotated operations, *23*

$\widehat{\beta}$: augmented environment for variables, *23*

$\widehat{\sigma}$: augmented environment for defined functions, *23*

$\widehat{\mathfrak{M}}[\![e]\!](\widehat{\beta}, \widehat{\sigma})$: augmented semantics of expression e, *23*

$\widehat{\mathfrak{M}}[\![P]\!]$: augmented semantics of program P, *24*

$\Phi_{\widehat{\mathfrak{M}},P}$: augmented semantic transformation for program P, *24*

$\twoheadleftarrow$: sort dependence relation, *38*

$\overset{\star}{\twoheadleftarrow}$: transitive and reflexive closure of $\twoheadleftarrow$, *38*

$[cs]$: equivalence class of cs wrt. $\overset{\star}{\twoheadleftarrow}$, *38*

$\mathsf{h}(cs)$: height of sort cs, *38*

A^t: abstract domains, *38*

$\preceq^t$: order on A^t, *38*

$\perp^t$: least element of A^t, *38*

$\leq_n$: bitwise less or equal on n–tuples, *39*

r_c: first recursive argument of constructor c, *41*

$\mathfrak{E}[\![c]\!]$: escape semantics of constructors, *41*

$\mathfrak{E}[\![\mathtt{sel}^k\text{–}c]\!]$: escape semantics of selectors, *42*

$\mathsf{papply}_m^{t_1,\ldots,t_n \to t}$: partial application of abstract functions, *44*

$\mathsf{apply}^{t_1,\ldots,t_n \to t}$: saturated application of abstract functions, *44*

$\mathfrak{E}[\![bf]\!]$: escape semantics of basic functions, *44*

$\mathfrak{E}[\![\mathtt{is}-c]\!]$: escape semantics of constructor tests, *45*

AOps: family of escape operations, *45*

χ: escape environment for variables, *45*

φ: escape environment for defined functions, *45*

$\mathfrak{E}[\![e]\!](\chi, \varphi)$: escape semantics of expression e, *45*

$\mathfrak{E}[\![P]\!]$: escape semantics of program P, *46*

$\Phi_{\mathfrak{E},P}$: escape semantic transformation for program P, *46*

α: abstraction function, *54*

HL: family of heap locations, *63*

free: allocation strategy, *63*

HN: heap nodes, *64*

$g_\emptyset$: empty heap function, *65*

$\rightarrowtail$: dependent locations, *65*

$\rightarrowtail^+$: transitive closure of $\rightarrowtail$, *65*

$\rightarrowtail^*$: transitive/reflexive closure of $\rightarrowtail$, *65*

$\overrightarrowtail{l}$: set of location on which l depends, *65*

$\overrightarrowtail{L}$: set of location on which any of L depends, *65*

rep: representation function, *66*

G: graph domain, *67*

$\preccurlyeq$: order on G, *67*

P^t: pointer domains, *68*

$\trianglelefteq^t$: order on P^t, *68*

NULL^t: least element of P^t, *68*

$G + hn$: graph resulting from allocation hn in G, *68*

$\mathfrak{G}[\![f]\!]$: graph semantics of intrinsic function f, *69*

GOps: family of graph operations, *70*

exec: execution function, *70*

ξ: graph environment for variables, *71*

θ: graph environment for defined functions, *71*

$\mathfrak{G}[\![e]\!](\xi, \theta)$: graph semantics of expression e, *71*

$\Phi_{\mathfrak{G},P}$: graph semantic transformation for program P, *73*

$\mathsf{FS}_{\mathfrak{G},P}$: function space for $\Phi_{\mathfrak{G},P}$, *73*

$\equiv_{\preccurlyeq}$: equivalence relation induced by $\preccurlyeq$, *80*

$\mathfrak{G}[\![P]\!]$: graph semantics of program P, *85*

$\aleph$: family of annotations, *96*

E_c: family of ctgc–annotated expressions, *96*

$G - l$: graph resulting from deleting the node at l in G, *97*

$G - L$: graph resulting from deleting all nodes of L in G, *97*

$G -_{a/L} l$: deallocation function, *97*

$\mathfrak{G}_c[\![e]\!](\xi, \theta)$: graph semantics of ctgc–annotated expression e, *99*

$\mathsf{abstoann}$: abstract values to annotations, *102*

$a_1 \simeq_f a_2$: functionally equivalent abstract values, *103*

$\mathsf{ctgc}_{\mathfrak{E}}$: ctgc–annotation of expressions based on $\mathfrak{E}$, *103*

SL: family of stack locations, *113*

C. Source Code for C Version of qs

Representation of `ListOfInt`

`intlist.h`:

```
typedef struct t_intlistnode *intlist;

typedef struct t_intlistnode {
  int     typ;
  int     entry;
  intlist next;
} intlistnode;

intlist NIL ();
intlist CONS(int a, intlist l);
void    FREE (intlist l);
```

`intlist.c`:

```
#include "malloc.h"
#include "stdio.h"
#include "intlist.h"

intlist NIL()
{
  intlist result;
  result = (intlist) malloc(sizeof(intlistnode));
  result->typ=0;
  result->next=NULL;
  return result;
}

intlist CONS(int a, intlist l)
{
  intlist result;
  result = (intlist) malloc(sizeof(intlistnode));
  result->typ=1;
  result->entry=a;
  result->next=l;
  return result;
}

void FREE(intlist l)
{
  intlist help;

  while (l->typ) {
```

```
    help=l->next;
    free(l);
    l=help;
  }
  free(l);
}
```

Implementation of ***append***

append.h:

```
intlist append (intlist l1, intlist l2);
```

append.c:

```
#include "intlist.h"
#include "append.h"

intlist append (intlist l1, intlist l2)
{
  intlist result;

  if (!l1->typ) {       /* NIL */
    result=l2;
  } else {             /* CONS */
    result=CONS(l1->entry,append(l1->next,l2));
  }
  return result;
}
```

Implementation of First–Order Versions of ***filter***

filter.h:

```
intlist filter_less   (int a, intlist l);
intlist filter_moreeq (int a, intlist l);
```

filter.c:

```
#include "intlist.h"
#include "filter.h"

intlist filter_less (int a, intlist l)
{
  intlist result;

  if (!l->typ) {        /* NIL */
    result=NIL();
  } else {              /* CONS */
    intlist help;
    help=filter_less(a, l->next);
    if (l->entry<a)
```

```
      result=CONS(l->entry,help);
    else
      result=help;
  }
  return result;
}

intlist filter_moreeq (int a, intlist l)
{
  intlist result;

  if (!l->typ) {      /* NIL */
   result=NIL();
 } else {             /* CONS */
   intlist help;
   help=filter_moreeq(a, l->next);
   if (l->entry>=a)
     result=CONS(l->entry,help);
   else
     result=help;
  }
  return result;
}
```

Implementation of `qs`

`qs.h`:

```
intlist qs (intlist l);
```

`qs.c` (Without CTGC):

```
#include "intlist.h"
#include "filter.h"
#include "append.h"
#include "qs.h"

intlist quicksort (intlist l)

{
  intlist result;

  if (!l->typ) {       /* NIL */
    result=NIL();
  } else {             /* CONS */
    result=append(qs(filter_less(l->entry, l->next)),
                  append(CONS(l->entry,NIL())),
                         qs(filter_moreeq(l->entry, l->next)));
  }
  return result;
}
```

`qs.c` (With CTGC):

```
#include "intlist.h"
#include "filter.h"
#include "append.h"
#include "qs.h"

intlist quicksort (intlist l)

{
  intlist result;

  if (!l->typ) {       /* NIL */
    result=NIL();
  } else {             /* CONS */
    intlist qsarg1, qsarg2, aparg1, aparg2, aparg3, help;

    qsarg1=filter_less(l->entry, l->next);
    aparg1=quicksort(qsarg1);
    FREE(qsarg1);

    help  =CONS(l->entry,NIL());
    aparg3=append(aparg1,help);
    FREE(aparg1);

    qsarg2=filter_moreeq(l->entry, l->next);
    aparg2=quicksort(qsarg2);
    FREE(qsarg2);

    result=append(aparg3,aparg2);
    FREE(aparg3);
  }

  return result;
}
```

Bibliography

[Abr86] S. Abramsky. Strictness Analysis and Polymorphic Invariance. In Ganzinger and Jones [GJ86], pages 1–23.

[AH87a] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[AH87b] S. Abramsky and C. Hankin. An Introduction to Abstract Interpretation. In *Abstract Interpretation of Declarative Languages* [AH87a], chapter 1, pages 63–102.

[AP95] P. Achten and R. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81—110, January 1995.

[Arg89] G. Argo. Improving the Three Instruction Machine. In FPCA'89 [FPC89], pages 100–115.

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Science, Chalmers University, Sweden, 1987.

[Ban97] A. Banerjee. A Modular, Polyvariant, and Type–Based Closure Analysis. In *Proceedings of the 1997 International Conference on Functional Programming (ICFP)*, pages 1—10. ACM, June 1997.

[BD77] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BHA86a] G. L. Burn, C. Hankin, and S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7:249–278, 1986.

[BHA86b] G. L. Burn, C. Hankin, and S. Abramsky. The Theory of Strictness Analysis for Higher Order Functions. In Ganzinger and Jones [GJ86], pages 42–62.

[BMC94] C. Baier and M. E. Majster-Cederbaum. Denotational semantics in the cpo and metric approach. *Theoretical Computer Science*, 135(2):171–220, December 1994. Fundamental Study.

[Boe91] H. Boehm. Space efficient conservative garbage collection. In *Proc. of the ACM SIGPLAN '91 Conference*, number 28 in SIGPLAN Notices, 1991.

[BS95] E. Barendsen and S. Smetsers. Uniqueness Type Inference. In Hermenegildo and Swierstra [HS95b], pages 189—206.

[BTV96] L. Birkedal, M. Tofte, and M. Vejlstrup. From Region Inference to von Neumann Machines via Region Representation Inference. In *Proceedings of the 23th Symposium on Principles of Programming Languages (POPL)*, pages 171–183. ACM, January 1996.

[BvEG+87] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term Graph Rewriting. In J. W. de Bakker, A. J.

Nijman, and P. C. Treleaven, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE) (Volume 2: Parallel Languages)*, number 259 in Lecture Notes in Computer Science, pages 141–158. Springer–Verlag, 1987.

[BvELP87] T. Brus, M. van Ecklen, M. Van Leer, and M. Plasmeijer. Clean – A Language for Functional Graph Rewriting. In Kahn [Kah87], pages 364–384.

[BW88] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience*, pages 807–820, September 1988.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points. In *Proceedings of the 4th Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, January 1977.

[Chi94] W. Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, October 1994.

[Cou97] P. Cousot. Types as Abstract Interpretations. In POPL'97 [POP97], pages 316–331.

[dBZ82] J. W. de Bakker and J. I. Zucker. Processes and the Denotational Semantics of Concurrency. *Information and Control*, 54(1/2):70–120, July 1982.

[Deu97] A. Deutsch. On the Complexity of Escape Analysis. In POPL'97 [POP97], pages 358–371.

[Dup95] L. Dupré. *BUGS in Writing: A Guide to Debugging Your Prose.* Addison–Wesley, 1995.

[Fax95] K. Faxén. Optimising Lazy Functional Programs Using Flow Inference. In Mycroft [Myc95], pages 136—153.

[FB93] S. Finne and G. Burn. Assessing the Evaluation Transformer Model of Reduction on the Spineless G-machine. In FPCA'93 [FPC93], pages 331–340.

[Fla97] D. Flanagan. *Java in a Nutshell.* O'Reilly, second edition, 1997.

[FPC89] *Proceedings of the 4th Conference on Functional Programming Languages and Computer Architecture (FPCA).* ACM, September 1989.

[FPC93] *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture (FPCA).* ACM, September 1993.

[FW87] J. Fairbairn and S. Wray. Tim: A Simple, Lazy Abstract Machine to Execute Supercombinators. In Kahn [Kah87], pages 34–46.

[FW88] A. B. Ferguson and P. Wadler. When will deforestation stop? In *Functional Programming, Glasgow 1988. Workshops in Computing*, pages 39–56, 1988.

[GJ86] H. Ganzinger and N. Jones, editors. *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science. Springer–Verlag, 1986.

[GLP93] A. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. In FPCA'93 [FPC93].

[GP90] B. Goldberg and Y. G. Park. Higher Order Escape Analysis: Optimizing Stack Allocation in Functional Program Implementations. In Jones [Jon90], pages 152–160.

[Grä71] G. Grätzer. *Lattice Theory: First Conzepts and Distributive Lattices.* W. H. Freeman, 1971.

[Gun94] C. A. Gunter. Semantics of Types in Programming Languages. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science Volume 3: Semantic Structures*, pages 399–476. Oxford University Press, 1994.

[Ham95] G. W. Hamilton. Compile–Time Garbage Collection for Lazy Functional Languages. In H. G. Baker, editor, *Memory Management, Proccedings of the International Workshop IWMM 95*, number 986 in Lecture Notes in Computer Science, pages 199—144. Springer–Verlag, 1995.

[Ham96] G. W. Hamilton. Higher Order Deforestation. In H. Kuchen and S. Doaitse Swierstra, editors, *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP)*, number 1140 in Lecture Notes in Computer Science, pages 213—227. Springer–Verlag, 1996.

[HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving Structural Hylomorphisms From Recursive Definitions. In *Proceedings of the 1996 International Conference on Functional Programming (ICFP)*, pages 73–82. ACM, June 1996.

[HJ90] G. W. Hamilton and S. B. Jones. Compile-Time Garbage Collection by Necessity Analysis. In Peyton Jones et al. [PHK90], pages 66–70.

[HS95a] F. Henglein and D. Sands. A Semantic Model of Binding Times for Safe Partial Evaluation. In Hermenegildo and Swierstra [HS95b].

[HS95b] M. Hermenegildo and S. Doaitse Swierstra, editors. *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP)*, number 982 in Lecture Notes in Computer Science. Springer–Verlag, 1995.

[Hug92] S. Hughes. Compile-time Garbage Collection for Higher-Order Functional Languages. *Journal of Logic and Computation*, 2(4):483–509, 1992.

[ISY88] K. Inoue, H. Seki, and H. Yagi. Analysis of Functional Programs to Detect Run-Time Garbage Cells. *TOPLAS*, 10(4):555–578, October 1988.

[JL96] R. Jones and R. Lins. *Garbage Collection.* Wiley, 1996.

[JM81] N. D. Jones and S. S. Muchnick. Complexity of Flow Analysis, Inductive Assertion Synthesis and a Language Due to Dijkstra. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 12. Prentice–Hall, 1981.

[JM89] S. B. Jones and D. Le Métayer. Compile-Time Garbage Collection by Sharing Analysis. In FPCA'89 [FPC89], pages 54–74.

[JM90] T. P. Jensen and T.Æ. Mogensen. A Backward Analysis for Compile-time Garbage Collection. In Jones [Jon90], pages 227–239.

[Joh87] T. Johnsson. *Compiling Lazy Functional Languages, Part I.* PhD thesis, Department of Computer Science, Chalmers University, Sweden, 1987.

[Jon90] N. Jones, editor. *Proceedings of the 3rd European Symposium on Programming (ESOP)*, number 432 in Lecture Notes in Computer Science. Springer–Verlag, 1990.

[Jon95] S. B. Jones. An Experiment in Compile Time Garbage Collection. Technical Report 84, Programming Methodology Group, University of Goteborg and Chalmers University, 1995.

[JW90] S. B. Jones and M. White. Is Compile Time Garbage Collection Worth The Effort? In Peyton Jones et al. [PHK90], pages 172–176.

[Kah87] G. Kahn, editor. *Proceedings of the 3rd Conference on Functional Programming Languages and Computer Architecture (FPCA)*, number 274 in Lecture Notes in Computer Science. Springer–Verlag, September 1987.

[Knu77] D. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

[LS92] J. Launchbury and P. Sansom, editors. *Functional Programming, Glasgow, 1992*. Springer–Verlag, 1992.

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley, 1997.

[MCB96] M. E. Majster-Cederbaum and C. Baier. Metric completion versus ideal completion. *Theoretical Computer Science*, 170(1–2):145–171, December 1996.

[Moh] M. Mohnen. Escape Analysis Revisited. Submitted for ICCL 1998.

[Moh95a] M. Mohnen. Efficient Closure Utilisation by Higher-Order Inheritance Analysis. In Mycroft [Myc95], pages 261–278.

[Moh95b] M. Mohnen. Efficient Compile-Time Garbage Collection for Arbitrary Data Structures. In Hermenegildo and Swierstra [HS95b], pages 241–258.

[Moh95c] M. Mohnen. `FUNC`: A Functional Language ... and more! In T. Johnsson, editor, *Proceedings of the 7th International Workshop on Implementation of Functional Languages (IFL)*, Tech. Report, pages 134–140. Chalmers University of Technology, 1995.

[Moh95d] M. Mohnen. Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages. Technical Report AIB-95-09, RWTH Aachen, 1995.

[Moh97] M. Mohnen. Using Quasi Ordered Sets to Model Program Properties Denotationally. In M. Jahre, K. Pasedach, and K. Pohl, editors, *Informatik '97: Informatik als Innovationsmotor*, Informaik aktuell, pages 532–541. Springer–Verlag, 1997.

[MW92] S. Marlow and P. Wadler. Deforestation for Higher–Order Functions. In Launchbury and Sansom [LS92], pages 66–70.

[Myc80] A. Mycroft. The Theory and Practice of Transforming Call-by-Need into Call-by-Value. In *Proceedings of the International Symposium on Programming*, number 83 in LNCS, pages 269–281. Springer–Verlag, 1980.

[Myc95] A. Mycroft, editor. *Proceedings of the 2nd International Symposium on Static Analysis (SAS)*, number 983 in Lecture Notes in Computer Science. Springer–Verlag, 1995.

[NN92a] F. Nielson and H. R. Nielson. Finiteness Conditions for Fixed Point Iteration. In J. L. White, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming (LaFP)*, pages 96–108. ACM, 1992.

[NN92b] H. R. Nielson and F. Nielson. Bounded Fixed Point Iteration. In POPL'92 [POP92], pages 71–82.

[NN92c] H. R. Nielson and F. Nielson. Bounded Fixed-Point Iteration. *Journal of Logic and Computation*, 2(4):441–464, August 1992.

[Par92] W. Partain. The **nofib** Benchmark Suite of Haskell Programs. In Launchbury and Sansom [LS92], pages 195–202.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Pey92] S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PG92] Y. G. Park and B. Goldberg. Escape Analysis on Lists. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN Notices 27(7), pages 116–127. ACM, June 1992.

[PHK90] S. L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors. *Functional Programming, Glasgow, 1990.* Springer–Verlag, 1990.

[PL91a] S. Peyton Jones and D. R. Lester. *Implementing Functional Languages: A Tutorial.* Prentice–Hall, 1991.

[PL91b] S. L. Peyton Jones and J. Launchbury. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In R. J. M. Hughes, editor, *Proceedings of the 5th Conference on Functional Programming Languages and Computer Architecture (FPCA)*, number 523 in Lecture Notes in Computer Science, pages 636–666. Springer–Verlag, 1991.

[POP92] *Proceedings of the 19th Symposium on Principles of Programming Languages (POPL).* ACM, January 1992.

[POP97] *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL).* ACM, January 1997.

[PP93] S. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In J. T. O'Donnell and K. Hammond, editors, *Functional Programming, Glasgow, 1993*, pages 66–70. Springer–Verlag, 1993.

[PS89] S. L. Peyton Jones and J. Salkild. The Spineless Tagless G-Machine. In FPCA'89 [FPC89], pages 184–201.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[PW92] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In ACM [POP92], pages 71–84.

[RP90] J. C. Reynolds and G. D. Plotkin. On Functors Expressible in the Polymorphic Lambda Calculus. In G. Huet, editor, *Logical Foundations of Functional Programming*, University of Texas at Austin Year of Programming Series. Addison–Wesley, 1990. Submitted to Information and Computation. Also available as CMU School of Computer Science technical report number CMU-CS-90-147.

[San95] A. Santos. *Compilation by Transformation in Non–Strict Functional Languages.* PhD thesis, University of Glasgow, 1995.

[SBvEP94] S. Smetsers, E. Barendsen, M. van Eekelen, and R. Plasmeijer. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In H. J. Schneider and H. Ehrig, editors, *Proc. of Graph Transformations in Computers Science, International Workshop, Dagstuhl Castle, Germany*, number 776 in LNCS, pages 358–379. Springer–Verlag, 1994.

[See87] R. A. G. Seely. Categorical semantics for higher order polymorphic lambda calclus. *Journal of Symbolic Logic*, 52, December 1987.

[Sei96] H. Seidl. Integer Constraints to Stop Deforestation. In H. R. Nielson, editor, *Proceedings of the 6th European Symposium on Programming (ESOP)*, number 1058 in Lecture Notes in Computer Science. Springer–Verlag, 1996.

[Ses91] P Sestoft. *Analysis and efficient implementation of functional programs.* PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[Shi91] O. G. Shivers. *Control–Flow Analysis of Higher–Order Languages or Taming Lambda.* PhD thesis, Carnige–Mellon Univeristy, May 1991.

[SHLG94] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor. *Mathematical Theory of Domains.* Number 21 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

[Sør94] M. H. Sørensen. A Grammar-Based Data-Flow Analysis to Stop Deforestation. In S. Tison, editor, *Proceedings of the 19th Colloquium on Trees in Algebra and Programming (CAAP)*, number 787 in Lecture Notes in Computer Science, pages 335–351. Springer–Verlag, 1994.

[SS97] H. Seidl and M. H. Sørensen. Constraints to Stop Higher–Order Deforestation. In POPL'97 [POP97], pages 400—413.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Thi94] P. Thiemann. *Grundlagen der funktionalen Programmierung.* Teubner, 1994. (in German).

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proceedings of the 7th Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 306–313. ACM, September 1995.

[Tof95] M. Tofte. Region Inference for Higher–Order Functional Languages. In Mycroft [Myc95].

[TT94] M. Tofte and J.-P. Talpin. Implementation of the Typed Call-by-Value λ–calculus using a Stack of Regions. In *Proceedings of the 21th Symposium on Principles of Programming Languages (POPL)*, pages 188–201. ACM, January 1994.

[VWL94] B. Vergauwen, J. Wauman, and J. Lewi. Efficient FixPoint Computation. In B. Le Charlier, editor, *Proceedings of the 1st International Symposium on Static Analysis (SAS)*, number 864 in Lecture Notes in Computer Science, pages 314–328. Springer–Verlag, 1994.

[Wad71] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus.* Ph.D. thesis, Programming Research Group, Oxford University, September 1971.

[Wad84] P. Wadler. Listlessness is Better than Laziness. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming (LaFP)*, pages 45–52. ACM, August 1984.

[Wad86] P. Wadler. Listlessness is better than laziness II: composing listless functions. In Ganzinger and Jones [GJ86].

[Wad87] P. Wadler. Strictness analysis on non–flat domains. In Abramsky and Hankin [AH87a], chapter 12, pages 266–275.

[Wad90] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.

[Wec92] W. Wechler. *Universal Algebra for Computer Scientists.* Number 25 in EATCS Monographs of Theoretical Computer Science. Springer–Verlag, 1992.

[Wel84] T. A. Welch. A Technique for High-Performance Data Compression. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society*, 17(6), June 1984.

Bildungsgang

Name:	Markus Mohnen
geboren am:	24.09.67
in:	Erkelenz

1974–78	Besuch der Grundschule in Wassenberg
1978–87	Besuch des Gymnasiums der Stadt Hückelhoven mit Abschluß Abitur im Juni 1987
Okt. 1987	Beginn des Studiums der Informatik und Mathematik an der Rheinisch–Westfälischen Technischen Hochschule Aachen
Okt. 1989	Vordiplom in Informatik mit Nebenfach Mathematik
Sep. 1991	Vordiplom in Mathematik mit Nebenfach Informatik
Jan. 1993	Beendigung des Studiums mit Abschluß Diplom in Informatik und Nebenfach Mathematik
Feb. 1993	Wissenschaftliche Hilfskraft am Lehrstuhl für Informatik II der RWTH Aachen
seit April 1993	Wissenschaftlicher Angestellter am Lehrstuhl für Informatik II der RWTH Aachen