Type-based amortized stack memory prediction

Brian Campbell

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2007

Abstract

Controlling resource usage is important for the reliability, efficiency and security of software systems. Automated analyses for bounding resource usage can be invaluable tools for ensuring these properties.

Hofmann and Jost have developed an automated static analysis for finding linear heap space bounds in terms of the input size for programs in a simple functional programming language. Memory requirements are amortized by representing them as a requirement for an abstract quantity, *potential*, which is supplied by assigning potential to data structures in proportion to their size. This assignment is represented by annotations on their types. The type system then ensures that all potential requirements can be met from the original input's potential if a set of linear constraints can be solved. Linear programming can optimise this amount of potential subject to the constraints, yielding a upper bound on the memory requirements.

However, obtaining bounds on the heap space requirements does not detect a faulty or malicious program which uses excessive *stack* space.

In this thesis, we investigate extending Hofmann and Jost's techniques to infer bounds on stack space usage, first by examining two approaches: using the Hofmann-Jost analysis unchanged by applying a CPS transformation to the program being analysed, then showing that this predicts the stack space requirements of the original program; and directly adapting the analysis itself, which we will show is more practical.

We then consider how to deal with the different allocation patterns stack space usage presents. In particular, the temporary nature of stack allocation leads us to a system where we calculate the total potential after evaluating an expression in terms of assignments of potential to the variables appearing in the expression as well as the result. We also show that this analysis subsumes our previous systems, and improves upon them.

We further increase the precision of the bounds inferred by noting the importance of expressing stack memory bounds in terms of the depth of data structures and by taking the maximum of the usage bounds of subexpressions. We develop an analysis which uses richer definitions of the potential calculation to allow depth and maxima to be used, albeit with a more subtle inference process.

Acknowledgements

My thanks go to my supervisor, Don Sannella, for his advice and support, the Mobility and Security group, and the Laboratory for Foundations of Computer Science for being such a nice place to work and filled with interesting people. I would also like to thank Steffen Jost for our discussions on our respective work.

I am grateful for the support that my parents have provided me with over the last few years, including telling me to get on with my work, and to take time to relax — often concurrently.

This research was supported by the MRG project (IST-2001-33149) which was funded by the EC under the FET proactive initiative on Global Computing.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Brian Campbell*)

Table of Contents

Chapter 1

Introduction

Automatically predicting resource requirements for programs is of considerable interest. Such analyses can be used to prevent failures due to exhausted resources, discover 'hot spots' in programs where improvements would yield the greatest gain in efficiency, debug unexpectedly high resource usage, or check that malicious parties will be unable to consume and withhold resources. This is particularly important in highly constrained environments such as smart cards, especially where failures are difficult or expensive to recover from. Here we are primarily concerned with memory usage, although in principle the techniques can be applied for other resources.

Hofmann and Jost have presented an automatic linear heap space analysis for a functional programming language (Hofmann and Jost, 2003). It is an amortized analysis; data structures are assigned some amount of *potential* to 'pay' for later allocations, and changes in allocation are conservatively approximated by changes in potential. Thus the potential at the start of the program is a bound on the free memory required. The potential is represented by type annotations, and the type system ensures that it is sufficient for all allocations. However, their analysis does not include stack memory, so some forms of excessive memory consumption may go unnoticed. In fact, restricting ourselves to a linear heap size does not have a huge effect on the class of functions that can be evaluated; with an unbounded stack we can compute any of the class of functions requiring $O(2^{cn})$ time (Cook, 1971).

Stack memory is used in a different way to heap memory. It is usually short-lived; providing temporary information about the progress in processing a data structure, rather than forming a data structure in its own right. One important consequence is that the stack memory usage of a program is typically proportional to the *depth* of its input data structures, not their *total* sizes.

The thesis of this work is that type-based amortized analyses can be developed which provide good bounds on stack memory usage for programs which run in linear space. Thus we will be able to estimate the total memory usage of a program by combining the heap space analysis with one of our stack space analyses.

1.1 Outline

In Chapter 2 we introduce the simple LFD programming language which will be used to develop the analyses, and survey the Hofmann-Jost analysis for heap memory.

In Chapter 3 we consider using a form of continuation passing style (CPS) transformation to yield a new program which only uses heap space. Thus Hofmann-Jost can be used on the new program to predict total memory requirements. We then consider the effectiveness of this approach.

Then in Chapter 4 we provide a more direct analysis on the original program including a treatment of tail-call optimisation. We examine the limitations of this analysis.

In Chapter 5 we present an extension to overcome poor approximations of stack space caused by limitations on which data structure sizes the bounds are parametrised by. The soundness of the extended analysis (and the previous analyses) is proven.

In Chapters 6 and 7 our attention turns to using the depth of data structures in the bounds. We present a type system for the depth analysis in Chapter 6 which uses extra structure in the typing context to determine the form of the bounds, then prove its soundness and give some examples. Following this, in Chapter 7 we complete the depth analysis by providing an inference procedure for the type system.

In Chapter 8 we consider related work on the Hofmann-Jost system and other approaches to bounding and verifying resource usage. In Chapter 9 we consider further work that could be conducted based upon the analyses we have developed.

Finally, in Chapter 10 we present our conclusions.

We also give an extended example of our analyses (on a functional heap sort program) in Appendix A.

Chapter 2

Background

To develop our analyses we require a language with clear semantics to study, and an understanding of the Hofmann-Jost system that we base them upon. In this chapter we introduce the LFD language and its operational semantics (including metering of space usage), along with some variations of the semantics. We will also discuss some of the issues surrounding memory management in the language. Finally, we examine the Hofmann-Jost system for bounding heap memory requirements.

This chapter is primarily based on Hofmann and Jost's original paper (Hofmann and Jost, 2003). We augment the operational semantics with stack space usage metering, tail-call optimisation, and partial executions to allow reasoning about non-terminating programs. These features will be needed for later chapters. The analysis that we present in this chapter is extended by 'resource polymorphism', due to its importance for practical use and to discuss the effects of this polymorphism on the complexity of the analysis.

2.1 The LFD language

For consistency with Hofmann and Jost's work we use their basic language, LF, extended with algebraic datatypes instead of built-in lists. The resulting LFD language is close to the language used in Jost's implementation of their analysis (Jost, 2004b). We include algebraic datatypes because they are required for the transformation described in Chapter 3. It is a simple first-order call-by-value functional programming language.

We consider a first-order language in part because higher-order extensions of the Hofmann-Jost system are a topic of ongoing research by Jost, and also because their main impact from our perspective is on individual stack frame sizes, which we take as

$$
\begin{aligned}
P &:= \mathsf{let}\ B \mid \mathsf{let}\ B\ P \\
B &:= D \mid D\ \mathsf{and}\ B \\
D &:= f(x_1,\ldots,x_p) = e_f \\
e &:= * \mid \mathsf{true} \mid \mathsf{false} \mid x \mid f(x_1,\ldots,x_p) \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \\
&\quad \mid (x_1,x_2) \mid \mathsf{match}\ x\ \mathsf{with}\ (x_1,x_2) \to e \\
&\quad \mid \mathsf{inl}(x) \mid \mathsf{inr}(x) \mid \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r \\
&\quad \mid c(x_1,\ldots,x_p) \mid \mathsf{match}\ x\ \mathsf{with}\ p_1 \to e_1 \mid \cdots \mid p_m \to e_m \\
p &:= c(x_1,\ldots,x_p) \mid c(x_1,\ldots,x_p)'
\end{aligned}
$$

Figure 2.1: Syntax

given[1].

The syntax for programs is given in Figure 2.1, where f is a function name, $*$ is the value of unit type, c is a constructor name, x and x_i are variable names and the e_i are subexpressions. Programs, P, take the form of a number of function definitions, D, arranged in mutually recursive groups, B. We may omit the parenthesis from function calls and declarations when there is only one argument. Patterns, p, can be destructive or non-destructive ($'$) to provide memory management information. We will discuss the distinction shortly.

We will implicitly assume that all bound variable names and function names are unique throughout. There is no requirement for match expressions to be exhaustive. In fact, requiring the addition of 'dummy' cases which will never be executed can interfere with the analyses. Instead we allow the program to fail upon a bad match by not yielding a result. We do not rule out repeated patterns — while they are not particularly useful, it is worth noting that their presence does not affect the analyses, despite introducing non-determinism into the operational semantics.

The syntax requires the program to be in a 'let-normal' form by using variables rather than subexpressions where possible. This makes the evaluation order explicit and allows the typing rules to be simpler. It can be helpful to consider let-normal form as the intermediate language of a compiler. This is similar to K-normal form (Birkedal et al., 1996) and A-normal form (Flanagan et al., 1993) used in other analyses and compilers. Indeed, Jost's implementation (Jost, 2004b) uses a similar intermediate

[1] We will discuss this in a little more detail in Section 9.1.1.

language produced from the Camelot compiler as part of the Mobile Resource Guarantees project (Aspinall et al., 2005). There, the results of the analysis were used to produce machine-checkable certificates of heap space bounds.

The types are $T := 1 \mid \mathsf{bool} \mid T \otimes T \mid T + T \mid ty$ for unit, boolean, pairs, sums and algebraic datatypes, respectively. The *ty*s range over a set of opaque type names. Our analyses will annotate these types to indicate resource requirements. Function signatures are of the form $T_1, \ldots, T_p \to T$ and constructors have similar signatures $T_1, \ldots, T_p \to ty$, or just ty for a nullary constructor. Thus each constructor is associated with a unique type ty. We leave parametric polymorphism to future work, see Section 9.1.1. As a result, we omit type constructors such as generic lists to reduce the amount of notation.

The product and sum types could be subsumed into the algebraic datatypes, but are included here to provide a contrast between heap-allocated datatype values, and product and sum values which are not heap-allocated except where they are included in a datatype value. This distinction becomes particularly important in Chapter 5.

For the analyses we presume that the unannotated types have already been inferred. This can be done with standard unification based type inference.

To reason about heap space we require some mechanism to limit the lifetime of heap allocated data, so we mark places in the code where deallocations can safely occur. Only algebraic datatype values are heap allocated, so we distinguish between (potentially) destructive $c(\ldots)$ match cases and benign, 'read-only' $c(\ldots)'$ ones. There is more than one possible implementation of heap management using these marks. A direct approach is to explicitly perform deallocation when executing match expressions, although this may result in some memory fragmentation that we do not take into account. Alternatively, we could use compacting garbage collection, where the 'destructive' matches provide a conservative approximation of the value's lifetime.

This choice affects the meaning of the bounds which we infer. The total amount of *live* memory is always within the bound, and with immediate deallocation the total *allocated* memory respects the bound. For compacting garbage collection the memory used *after* collection respects the bound, or we may go further and use the bound as a trigger for collection and always remain under it (modulo any extra space required for the collection).

We presume the existence of some external analysis which ensures that the destructive marks are used safely so that no data can be deallocated while live references to it exist, a property called *benign sharing*. Aspinall and Hofmann's usage aspects (As-

pinall et al., 2008) or Konečný's DEEL typing (Konečný, 2003) are suitable systems. They also provide a conservative estimate of the set of variables whose values do not share any heap locations with the result of a given expression in the program. We will make use of this separation property in Chapter 5.

Example 2.1. Suppose we have a datatype boollist with constructors

$$\mathsf{nil} : \mathsf{boollist} \quad \text{and} \quad \mathsf{cons} : \mathsf{bool}, \mathsf{boollist} \to \mathsf{boollist}.$$

A simple example of a program in this language is a function to negate a list of booleans:

```
let notlist l = match l with nil' -> nil | cons(h,t)' ->
                  let hh = if h then false else true in
                  let tt = notlist t in
                    cons(hh,tt)
```

The function uses a non-destructive match expression so that the argument, `l`, is left intact. As a result, the function needs extra heap memory equal to the amount of space occupied by the argument. Using a destructive match instead would allow it to run without requiring extra heap space, but that would only be suitable if the input list is never used again. Regardless of the variant used, the function requires stack space proportional to the length of the argument.

2.1.1 Operational semantics

Values $v \in \mathsf{val}$ in the operational semantics consist of unit, booleans, pairs, variants ($\mathsf{inl}(v)$ and $\mathsf{inr}(v)$ for a value v) and heap locations $l \in \mathsf{loc}$ for algebraic datatypes. The set of locations, loc, is assumed to be infinite and have a special location null which can represent one nullary constructor per datatype, for example, a nil list. We define a set nullc of the nullary constructors which are represented by null.

An environment S maps variables to values, and a store $\sigma \in \mathsf{heap}$ is a partial map from non-null locations to constructor and value tuples for the contents of each datatype value.

The operational semantics is given in Figures 2.2 and 2.3, with judgements of the form

$$m, S, \sigma \vdash e \rightsquigarrow v, \sigma', m'$$

meaning that with $m \in \mathbb{N}$ units of free memory, the environment S and the store σ, the expression e can be evaluated to value v, with the new store σ' and m' units of free memory. The evaluation of a whole program is realised by the evaluation of a chosen 'initial' function $f(x_1,\ldots,x_p)$ on some arguments provided as the values $v_1,\ldots,v_p$ and initial store σ,

$$m,[x_1 \mapsto v_1,\ldots,x_p \mapsto v_p],\sigma \vdash f(x_1,\ldots,x_p) \rightsquigarrow v,\sigma',m'.$$

The operational semantics uses two auxiliary functions to define the memory requirements. The first, size(c), gives the amount of heap memory required to store $c(v_1,\ldots,v_p)$ where each v_i is of the corresponding type T_i from c's signature. Thus, we assume that all values of the same type are allocated the same amount of space. Note that this requirement forces values of sum types to be assigned the same size regardless of the choice made at runtime. Algebraic datatypes are slightly different: the value is the *location* pointing to the data structure, and locations are always the same size. When $c \in \mathsf{nullc}$ we have size$(c) = 0$ because no memory is required for a constructor represented by the null location. The second auxiliary function, stack(f), gives the size of stack frame required to call function f.

The size and stack functions can be defined using concrete values from a particular compiler, yielding a precise account of memory use. However, we can also use simpler definitions to obtain rougher estimates. For example, we could obtain an estimate of stack(f) by examining the local variables in the function body of f. Such an estimate may be suitable for a variety of compilers. In many of our examples we will take the even simpler approach of assigning uniform sizes—essentially counting the number of objects or stack frames rather than their exact sizes.

Heap space can be considered alone, without regard for stack space, by fixing stack(f) to be zero everywhere. Similarly, stack space can be measured alone by fixing size$(c) = 0$ for all constructors c.

We will also require an unmetered form of the operational semantics, where the resource amounts are dropped from all of the rules. Judgements then take the form $S,\sigma \vdash e \rightsquigarrow v,\sigma'$. This is equivalent to setting *both* stack and size to be zero everywhere.

Finally, we need to formalise the guarantees that we expect a benign sharing analysis to give. First we define a reachability function $\mathcal{R}$ which gives the set of heap

$$\frac{}{m,S,\sigma \vdash * \rightsquigarrow *,\sigma,m}\ \text{(E-UNIT)} \qquad \frac{c \in \{\mathsf{true},\mathsf{false}\}}{m,S,\sigma \vdash c \rightsquigarrow c,\sigma,m}\ \text{(E-BOOL)}$$

$$\frac{}{m,S,\sigma \vdash x \rightsquigarrow S(x),\sigma,m}\ \text{(E-VAR)}$$

$$\frac{S(x_1) = v_1\ \ldots\ S(x_p) = v_p \qquad m,[y_1 \mapsto v_1,\ldots,y_p \mapsto v_p],\sigma \vdash e_f \rightsquigarrow v,\sigma',m' \qquad \text{the } y_i \text{ are the symbolic arguments in the definition of } f}{m + \text{stack}(f),S,\sigma \vdash f(x_1,\ldots,x_p) \rightsquigarrow v,\sigma',m' + \text{stack}(f)}\ \text{(E-FUN)}$$

$$\frac{m,S,\sigma \vdash e_1 \rightsquigarrow v_0,\sigma_0,m_0 \qquad m_0,S[x \mapsto v_0],\sigma_0 \vdash e_2 \rightsquigarrow v,\sigma',m'}{m,S,\sigma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v,\sigma',m'}\ \text{(E-LET)}$$

$$\frac{S(x) = \mathsf{true} \qquad m,S,\sigma \vdash e_t \rightsquigarrow v,\sigma',m'}{m,S,\sigma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \rightsquigarrow v,\sigma',m'}\ \text{(E-IFTRUE)}$$

$$\frac{S(x) = \mathsf{false} \qquad m,S,\sigma \vdash e_f \rightsquigarrow v,\sigma',m'}{m,S,\sigma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \rightsquigarrow v,\sigma',m'}\ \text{(E-IFFALSE)}$$

$$\frac{v = (S(x_1),S(x_2))}{m,S,\sigma \vdash (x_1,x_2) \rightsquigarrow v,\sigma,m}\ \text{(E-PAIR)}$$

$$\frac{S(x) = (v_1,v_2) \qquad m,S[x_1 \mapsto v_1][x_2 \mapsto v_2],\sigma \vdash e \rightsquigarrow v,\sigma',m'}{m,S,\sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ (x_1,x_2) \to e \rightsquigarrow v,\sigma',m'}\ \text{(E-MATCHPAIR)}$$

$$\frac{S(x) = v}{m,S,\sigma \vdash \mathsf{inl}(x) \rightsquigarrow \mathsf{inl}(v),\sigma,m}\ \text{(E-INL)} \qquad \frac{S(x) = v}{m,S,\sigma \vdash \mathsf{inr}(x) \rightsquigarrow \mathsf{inr}(v),\sigma,m}\ \text{(E-INR)}$$

$$\frac{S(x) = \mathsf{inl}(v_0) \qquad m,S[x_l \mapsto v_0],\sigma \vdash e_l \rightsquigarrow v,\sigma',m'}{m,S,\sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r \rightsquigarrow v,\sigma',m'}\ \text{(E-MATCHINL)}$$

$$\frac{S(x) = \mathsf{inr}(v_0) \qquad m,S[x_r \mapsto v_0],\sigma \vdash e_r \rightsquigarrow v,\sigma',m'}{m,S,\sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r \rightsquigarrow v,\sigma',m'}\ \text{(E-MATCHINR)}$$

Figure 2.2: Operational semantics

$$\frac{s = (c, S(x_1), \ldots, S(x_p)) \qquad c \notin \mathsf{nullc} \qquad l \notin \mathrm{dom}(\sigma)}{m + \mathrm{size}(c), S, \sigma \vdash c(x_1, \ldots, x_p) \rightsquigarrow l, \sigma[l \mapsto s], m} \text{ (E-CONSTRUCT)}$$

$$\frac{c \in \mathsf{nullc}}{m, S, \sigma \vdash c \rightsquigarrow \mathsf{null}, \sigma, m} \text{ (E-CONSTRUCTN)}$$

$$\frac{\begin{array}{c} S(x) = l \qquad \sigma(l) = (c_i, v_1, \ldots, v_p) \\ m + \mathrm{size}(c_i), S[x_1 \mapsto v_1, \ldots, x_p \mapsto v_p], \sigma \backslash l \vdash e_i \rightsquigarrow v, \sigma', m' \end{array}}{m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with} \cdots \mid c_i(x_1, \ldots, x_p) \rightarrow e_i \mid \cdots \rightsquigarrow v, \sigma', m'} \text{ (E-MATCH)}$$

$$\frac{\begin{array}{c} S(x) = l \qquad \sigma(l) = (c_i, v_1, \ldots, v_p) \\ m, S[x_1 \mapsto v_1, \ldots, x_p \mapsto v_p], \sigma \vdash e_i \rightsquigarrow v, \sigma', m' \end{array}}{m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \cdots \mid c_i(x_1, \ldots, x_p)' \rightarrow e_i \mid \ldots \rightsquigarrow v, \sigma', m'} \text{ (E-MATCH}')$$

$$\frac{S(x) = \mathsf{null} \qquad c_i \in \mathsf{nullc} \qquad m, S, \sigma \vdash e_i \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \cdots \mid c_i \rightarrow e_i \mid \ldots \rightsquigarrow v, \sigma', m'} \text{ (E-MATCHN)}$$

$$\frac{S(x) = \mathsf{null} \qquad c_i \in \mathsf{nullc} \qquad m, S, \sigma \vdash e_i \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \cdots \mid c_i' \rightarrow e_i \mid \ldots \rightsquigarrow v, \sigma', m'} \text{ (E-MATCHN}')$$

Figure 2.3: Operational semantics (continued)

locations reachable from a given value or environment:

$$\begin{aligned} \mathcal{R}(\sigma, *) = \mathcal{R}(\sigma, \mathsf{true}) = \mathcal{R}(\sigma, \mathsf{false}) &= \emptyset \\ \mathcal{R}(\sigma, (v_1, v_2)) &= \mathcal{R}(\sigma, v_1) \cup \mathcal{R}(\sigma, v_2) \\ \mathcal{R}(\sigma, \mathsf{inl}(v)) = \mathcal{R}(\sigma, \mathsf{inr}(v)) &= \mathcal{R}(\sigma, v) \\ \mathcal{R}(\sigma, \mathsf{null}) &= \emptyset \\ \mathcal{R}(\sigma, l) &= \{l\} \cup \bigcup_i \mathcal{R}(\sigma, v_i) \quad \text{where } \sigma(l) = (c, v_1, \ldots, v_p) \\ \mathcal{R}(\sigma, S) &= \bigcup_{v \in \mathrm{dom}(S)} \mathcal{R}(\sigma, v) \end{aligned}$$

Definition 2.2. We say that an execution satisfies the *benign sharing conditions* when:

1. At every use of E-MATCH the 'dead' location should not be accessible from the 'live' variables that the subexpression may use,

$$l \notin \mathcal{R}(\sigma, S[x_1 \mapsto v_1, \ldots, x_p \mapsto v_p] \restriction \mathrm{FV}(e_i)), \tag{2.1}$$

and

2. at every use of E-LET the parts of the heap needed for e_2 should not be altered by match expressions in e_1,

$$\sigma \restriction \mathcal{R}(\sigma, S_{e_2}) = \sigma_0 \restriction \mathcal{R}(\sigma, S_{e_2}), \tag{2.2}$$

where $S_{e_2} = S \restriction (\mathrm{FV}(e_2) \setminus x)$.

The heap separation property can also be formalised. For any expression e in the program, the benign sharing analysis should provide a set of variables $V_e \subseteq \mathrm{FV}(e)$ that do not overlap with the result of e. More precisely:

Definition 2.3. Given an expression e in some program, we say that a set of variables V_e are *separate from the result* of a given evaluation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ when we have

$$\mathcal{R}(\sigma, S \restriction V_e) \cap \mathcal{R}(\sigma', v) = \emptyset.$$

In the typing rules presented in Chapter 5 we will assume that we can find suitable sets of variables which will satisfy this condition during any evaluation of the corresponding expression *within* an evaluation of the whole program. This allows an analysis providing these sets to use sharing information derived from other parts of the program.

Both of these properties may be derived from (for example) the correctness theorem of (Konečný, 2003).

2.1.2 Tail call optimisation

The operational semantics above does not reflect a common expectation in functional programming that a tail recursive call will not use extra stack space. Indeed, compilers may provide more general forms of tail call optimisation and there may be wide variation in practice.

We would like a flexible approach to obtaining conservative bounds so that we may adjust the system when considering different environments. To this end we split our modelling of tail calls into noting when we are in tail position, and deciding what effect this has on the stack consumption.

Tracking which expressions are in tail position is straightforward. For the operational semantics we add a boolean flag to the judgements indicating whether the current expression is in tail position, and mark the premises as appropriate. We also add the current function's name to the judgements to provide more information about whether

a tail call is possible. Thus we use $\vdash^{f,t}$ instead of $\vdash$, where f is the function name and t is true or false. Thanks to the use of let-normal form, only the E-LET rule and E-FUN rule have premises with different flags to the conclusion because only the E-LET rule can introduce a subexpression that is not in tail position:

$$\frac{S(x_1) = v_1 \ \ldots \ S(x_p) = v_p \qquad m, [y_1 \mapsto v_1, \ldots, y_p \mapsto v_p], \sigma \vdash^{f,\mathsf{true}} e_f \rightsquigarrow v, \sigma', m' \qquad \text{the } y_i \text{ are the symbolic arguments in the definition of } f}{m + \mathrm{stack}'(g,f,t), S, \sigma \vdash^{g,t} f(x_1, \ldots, x_p) \rightsquigarrow v, \sigma', m' + \mathrm{stack}'(g,f,t)} \text{(E-FUN-TAIL)}$$

$$\frac{m, S, \sigma \vdash^{f,\mathsf{false}} e_1 \rightsquigarrow v_0, \sigma_0, m_0 \qquad m_0, S[x \mapsto v_0], \sigma_0 \vdash^{f,t} e_2 \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash^{f,t} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v, \sigma', m'} \text{(E-LET-TAIL)}$$

and the other rules merely propagate the flag to the subexpression, if there is one.

We replace the stack function with a stack$'$ function which also depends upon the calling function and tail position flag. Hence $\mathrm{stack}'(f,g,\mathsf{true})$ is the amount of stack memory required for a call from f to g in tail position. General tail call optimisation can be modelled by setting

$$\mathrm{stack}'(f,g,\mathsf{true}) = \mathrm{stack}(g) - \mathrm{stack}(f) \quad \text{and} \quad \mathrm{stack}'(f,g,\mathsf{false}) = \mathrm{stack}(g)$$

for all f, g. Note that this means that the stack space used may fall if we tail-call a function with a smaller frame size. We could use other definitions for stack$'$, such as restricting tail call optimisation to recursive function calls.

Finally, the evaluation judgement for the initial function (f, say) always requires a frame to be allocated,

$$m, [x_1 \mapsto v_1, \ldots, x_p \mapsto v_p], \sigma \vdash^{\mathrm{initial},\mathsf{false}} f(x_1, \ldots, x_p) \rightsquigarrow v, \sigma', m',$$

with the assumption that $\mathrm{stack}'(\mathrm{initial}, f, t) = \mathrm{stack}(f)$ for all f, t.

2.1.3 Partial executions

The big-step operational semantics above does not allow for non-terminating programs. Any judgement describing the evaluation of an expression e must end with some result v:

$$m, S, \sigma \vdash e \rightsquigarrow v, \sigma', m'.$$

Nevertheless, non-terminating programs are of considerable interest. For example, programs providing network services often run continuously and it is important for reliability that they do not leak memory.

$$\frac{}{m, S, \sigma \vdash e \rightsquigarrow \mathsf{halted}, \sigma, m} \quad \text{(E-STOP)}$$

$$\frac{m, S, \sigma \vdash e_1 \rightsquigarrow \mathsf{halted}, \sigma_0, m_0}{m, S, \sigma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{halted}, \sigma_0, m_0} \quad \text{(E-STOPLET)}$$

$$\frac{m, S, \sigma \vdash e_1 \rightsquigarrow v_0, \sigma_0, m_0 \qquad v_0 \neq \mathsf{halted} \qquad m_0, S[x \mapsto v_0], \sigma_0 \vdash e_2 \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v, \sigma', m'} \quad \text{(E-LET)}$$

Figure 2.4: Changes to the operational semantics for partial executions of programs

Thus we augment the semantics to allow the execution to be halted at any point. If a memory bound can be shown for all such 'partial' executions of a program then the program will run indefinitely without exceeding the bound.

We add two new rules to the operational semantics and modify E-LET slightly. These rules are given in Figure 2.4. The set of values is extended by the special value halted, to indicate that execution was prematurely terminated. The E-STOP rule says that we may halt execution at any time. The new precondition $v_0 \neq \mathsf{halted}$ on E-LET prevents any further execution, instead the new E-STOPLET rule propagates the halted value. E-LET is the only existing rule that requires modification because the semantics requires programs to be in let-normal form.

These changes also provide the semantics for programs with inexhaustive matches which fail, up to the evaluation of the match expression which fails. We will be able to show that our inferred memory bounds are respected despite the failure.

We will not make any claims about the amount of free memory at the point the program is halted, although with some care it should be possible to show that a bound on it can be extracted from the analyses.

2.2 The Hofmann-Jost analysis

Now we consider the analysis for bounding heap space usage developed by Hofmann and Jost (Hofmann and Jost, 2003). This work grew out of the study of programming languages which capture particular complexity classes. A simple method to construct these languages is to impose severe syntactic constraints which exclude 'expensive' programs. For example, (Bellantoni and Cook, 1992) describes limiting the use of recursion and the results of recursive functions to obtain a programming language which

characterises PTIME.

To construct more palatable languages for programming more features are required, such as inductive datatypes and higher-order functions, and we should use more natural restrictions to maintain the desired complexity class. See (Hofmann, 2000a) for a survey of construction techniques.

Inductive datatypes can be included in a restricted language by requiring all functions to be *non-size-increasing*, by which we mean that the result of the function is no larger than its arguments. Hofmann produced languages with non-size-increasing functions by using a linear type system with a special resource type, $\diamond$, inhabited by a singleton value also denoted $\diamond$. These values are obtained upon matching data structures, and are required to construct new ones. For instance, the cons constructor for integer lists might be given the signature

$$\mathsf{cons} : \mathsf{int} \times \diamond \times \mathsf{list} \to \mathsf{list},$$

instead of the usual $\mathsf{int} \times \mathsf{list} \to \mathsf{list}$. The linearity of the type system prevents multiple uses of $\diamond$ values and the data structures they were obtained from, so that the size of the result values is bounded by the size of the input arguments.

With structural recursion the resulting language characterises PTIME, or PSPACE if the linearity of some higher-order functions is relaxed slightly (Hofmann, 2003; Hofmann, 2002). Allowing full recursion yields a more typical functional programming language at the 'expense' of increasing the complexity class to EXPTIME (Hofmann, 2002). The first-order fragment, LFPL, is more interesting: the complexity is reduced[2] to $O(2^{cn})$ for constant c, and it can be implemented by a translation to C which *reuses* the memory from matched data structures rather than calling the normal allocator (Hofmann, 2000b). Values of the $\diamond$ type correspond to free memory cells and can be realised as pointers in the implementation. Now the linearity of the type system ensures memory safety. (Relaxing the linearity of non-$\diamond$ values in the type system can be used to construct memory safety analyses such as those discussed in Section 2.1.)

Thus, if we could take a program in a similar language without $\diamond$s and add them automatically, we would obtain a program which runs in heap space bounded by the number of $\diamond$s. Note that as well as reusing the space occupied by the original arguments, functions in these programs can obtain extra space by requiring $\diamond$ values as arguments. For example, if we have the following function which inserts an integer into a sorted list,

[2] In LFPL we cannot replace a data structure by a 'free' closure — a trick which increases the expressiveness of the higher-order language to EXPTIME.

```
let insert(e,l) =
  match l with nil -> cons(e,nil)
           | cons(h,t) ->
    if e < h then cons(e,cons(h,t))
             else cons(h,insert(e,t))
```

to make it a well-typed LFPL function we need to supply an extra $\diamond$ for the new list element:

```
let insert(e,d1,l) =
  match l with nil -> cons(e,d1,nil)
           | cons(h,d2,t) ->
    if e < h then cons(e,d1,cons(h,d2,t))
             else cons(h,d1,insert(e,d2,t))
```

$$\texttt{insert} : \mathsf{int} \times \diamond \times \mathsf{int\ list} \longrightarrow \mathsf{int\ list}.$$

The extra memory need not be a fixed amount. For instance, the function

```
let double l =
   match l with nil -> nil
        | cons(h,t) -> let t' = double t in cons(h,cons(h,t'))
```

$$\texttt{double} : \mathsf{bool\ list} \longrightarrow \mathsf{bool\ list}$$

duplicates each element in a list (assuming that the boolean `h` can be treated non-linearly). We would like to infer a new version

```
let double l =
  match l with nil -> nil
    | cons(h,d1,t) -> match h with (h',d2) ->
                        let t' = double t in cons(h,d1,cons(h,d2,t'))
```

$$\texttt{double} : (\mathsf{bool} \times \diamond)\ \mathsf{list} \longrightarrow \mathsf{bool\ list}$$

where we require an extra cell of memory for each input list element to allocate its duplicate. Counting the number of $\diamond$ values required gives a bound on the number of memory cells required by the original function.

Note that we do not need to traverse each data structure to count the $\diamond$s. Instead, we can derive a function from the type which maps the structure's size to the number

of $\diamond$ values. Then we can build a function for the entire signature and bound the heap memory requirements for the *original* function. The `double` example is simple:

$$\Upsilon_{\mathsf{bool}\times\diamond}(\cdot) = 1, \quad \text{so} \quad \Upsilon_{(\mathsf{bool}\times\diamond)\ \mathsf{list}}(n) = n, \quad \text{so} \quad \Upsilon_{\mathtt{double}}(n) = n.$$

That is, for a list of length n, `double` requires n extra cells of memory.

These functions can be considered as assigning *potential* to each data structure, in the sense of the 'physicist's view' of amortized analysis described by (Tarjan, 1985). In that work, differences between the real cost of an operation and the amortized cost are accounted for by changes in the potential of the data structure. On a cheap operation the potential can be increased, then on a complex operation (say, a tree rebalancing) the accumulated potential may be used to compensate for the extra time required. The potential can be interpreted as the amount of 'free time' you have spare to spend on later operations.

In our setting, the potential corresponds to free memory that we have for later operations. Allocation lowers potential (by consuming a $\diamond$ value), and deallocation increases potential (by providing a new $\diamond$ value).

The discrete $\diamond$ type can only represent integer amounts of memory of uniform size and requires explicit manipulation of $\diamond$ values to be introduced. The Hofmann-Jost analysis overcomes these limitations, replacing the $\diamond$ type with rational annotations on existing types to represent amounts of free memory.

2.2.1 Informal description

The Hofmann-Jost analysis also defines functions to assign potential to data structures. Numerical type annotations are used to derive these functions, rather than the presence of $\diamond$ types. As in LFPL, the analysis is based upon a type system which constrains these annotations to ensure that the resulting potential will be large enough to account for all allocations.

The system annotates typings and function signatures with non-negative rational values[3] in two places. First, we add 'before' and 'after' amounts to typing judgements and function signatures to represent fixed amounts of potential (free memory). The constraints on these will mirror the operational semantics, and so require the 'before' annotation at an allocation to be at least as large as the amount to be allocated plus the

[3]Fractional annotations can arise naturally in this system. For example, if we require a unit of memory for every second element of a boolean list l, then l will have the type $\mathsf{boollist}(1/2)$. See Example 2.8 on page 27.

'after' annotation. Similarly, when typing a deallocation such as

$$\mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil}\ \rightarrow e_1 \mid \mathsf{cons}(h,t) \rightarrow e_2,$$

the 'before' fixed amount for typing e_2 is higher than the amount for the whole match expression because the list cell can be reused.

Second, we place annotations on types to denote 'per-constructor' amounts of potential. So if a list x has type $\mathsf{boollist}(k)$ then the k annotation represents $k \times |x|$ units of potential, k for each cons. Where an annotated type appears in the context, its potential contributes to the bound on the memory that is sufficient for evaluation. The potential for an annotated result type is part of the lower bound on the amount of memory free after evaluation. In both cases we can calculate the total 'before' or 'after' bound by summing all of the potential from the types, plus the 'fixed' amount.

For example, consider the judgement

$$x : \mathsf{boollist}(k), n \vdash \mathsf{cons}(\mathsf{true}, x) : \mathsf{boollist}(k), n'$$

and assume that one unit of space is consumed when allocating a boolean list cell. From the context we see that we have $k \times |x|$ units of potential from x, plus the fixed amount, n. Afterwards we will have $k \times |\mathsf{cons}(\mathsf{true}, x)| + n' = k \times (|x| + 1) + n'$ units of potential, and we will have allocated one unit of space. Thus for a successful typing we require (after cancelling the $k \times |x|$)

$$n \geq 1 + k + n'. \tag{2.3}$$

The intuition behind this constraint is that we require one unit for allocation and reserve k units of potential for later processing of the new element in the list. When we use match to take an element from the list, we will 'release' the k units of potential again by adding them to the fixed amount (plus one unit more if we deallocate the element).

We add similar annotations to function signatures. For example, the `notlist` function might be given the function signature

$$\texttt{notlist} : \mathsf{boollist}(3), 0 \rightarrow \mathsf{boollist}(2), 0,$$

which says that if it is invoked with a boolean list x and $3 \times |x| + 0$ cells of memory are free, then all of the allocations in the function will succeed, and some boolean list y will be returned along with $2 \times |y| + 0$ free cells for later use. Note that this typing is not unique; we consider other values for the annotations below.

Allocating a constant sized data structure can transform a 'fixed' annotation into a 'per-constructor' one. For example, in

$$\cdot, 9 \vdash \mathsf{cons}(\mathsf{false}, \mathsf{cons}(\mathsf{false}, \mathsf{cons}(\mathsf{false}, \mathsf{nil}))) : \mathsf{boollist}(2), 0$$

we consume 3 cells for allocation, then the remaining $6 = 3 \times 2$ units of potential satisfy the 'per-cons' annotation of the list, 2.

To infer these annotations the typing rules give linear constraints that their values must satisfy, like Equation 2.3 above. We can use standard linear programming techniques such as the Simplex algorithm (Dantzig, 1963) to solve these constraints and find a minimal set of satisfying annotations. The 'objective function' should be chosen so as to minimise the annotations on the left hand side of the function signature, so that the bound is minimised. Note that we must analyse the whole program at once to obtain an optimal bound, although self-contained parts can be examined alone.

One subtlety is that we allow function signatures to have different values for annotations at each application outside of its definition. This *resource polymorphism* is required to reflect differing resource requirements at different points in the program. For example, consider the heap memory required in the following function:

```
let id l = let notl = notlist l in notlist notl
```

This allocates a fresh list for both applications of `notlist`. If the second is given the signature

$$\texttt{notlist} : \mathsf{boollist}(1), 0 \to \mathsf{boollist}(0), 0,$$

to indicate that it needs enough potential to make the new list but no more, then the type for the variable `notl` must be $\mathsf{boollist}(1)$. The expression `notlist l` must be given the same type, so the first use of `notlist` must be typed differently:

$$\texttt{notlist} : \mathsf{boollist}(2), 0 \to \mathsf{boollist}(1), 0.$$

Intuitively, this says that we need enough memory to allocate a new list *and* enough left over to satisfy the potential required for the `notlist notl` expression.

To allow these different typings, we use a set of abstract constraint variables for annotations rather than explicit rational values and collect the constraints. At each function application we make duplicate copies of the constraints for the function body with fresh constraint variables, except at applications involved in defining the function itself. A copy of the constraints is added to the function signature for this purpose. A set of satisfying assignments for the constraints in the initial function's signature gives

an upper bound on resource usage. This resource polymorphism extension was suggested in the conclusion to Hofmann and Jost's original paper, and which also features in Jost's later work on extensions to support higher-order functions (Jost, 2004a).

We can now give a more general signature for `notlist` of

$$\texttt{notlist} : \mathsf{boollist}(k), n \to \mathsf{boollist}(k'), n' \,|\, \{k \geq k' + 1,\ n \geq n'\},$$

and the signatures given above are particular solutions of the constraints. The constraints given in some of our examples are simplified; the full set generated by the type system will also contain constraints for annotations within the type derivation of the function's body. In a few examples we will just give a particular illustrative solution to the constraints, as with the earlier typings for `notlist`.

2.2.2 Formal definition

To formalise the system, we need a precise notion of the meaning of the annotations. The annotated types are

$$T_a := 1 \mid \mathsf{bool} \mid T_a \otimes T_a \mid (T_a, k_l) + (T_a, k_r) \mid ty(\overline{k}),$$

where k_l and k_r are constraint variables and $\overline{k}$ is a tuple of constraint variables. Sum types are annotated to reflect different resource requirements depending upon the choice made. Similarly, datatypes have different annotations for each constructor.

The constraints on annotations take the form of linear equalities and inequalities,

$$a_1 k_1 + \cdots + a_n k_n = a_{n+1} k_{n+1} + \cdots + a_m k_m + c, \text{ or}$$
$$a_1 k_1 + \cdots + a_n k_n \geq a_{n+1} k_{n+1} + \cdots + a_m k_m + c,$$

where $a_i \in \mathbb{Q}$ and $c \in \mathbb{Q}$. Thus a set of constraints, Φ, plus some objective function to optimise forms a linear program. In general we use inequalities when there is a need for weakening of a fixed amount of potential, and equalities elsewhere.

The function signatures now take the form

$$\Sigma(f) = T_1, \ldots, T_p, k \to T, k' \,|\, \Phi$$

where T_i and T are the annotated types for the arguments and the result respectively, k and k' are extra amounts of free memory required and released (analogous to n and n' above), and Φ is the set of constraints on annotations required for the body to typecheck. Constructor signatures have the form

$$\Sigma(c_i) = \forall \overline{k}. T_1, \ldots, T_p, k_i \to ty(\overline{k})$$

where $\overline{k}$ is a sequence of annotations that the type $ty(\overline{k})$ is quantified over. In general, $\overline{k}$ should include all the annotations in all of the constructor signatures associated with $ty(\overline{k})$.

In our examples we have introduced a datatype boollist. The constructors for this type now have the signatures

$$\Sigma(\mathsf{nil}) = \forall k_n, k_c.\ k_n \to \mathsf{boollist}(k_n, k_c)$$
$$\text{and} \quad \Sigma(\mathsf{cons}) = \forall k_n, k_c.\ \mathsf{bool}, \mathsf{boollist}(k_n, k_c), k_c \to \mathsf{boollist}(k_n, k_c).$$

The signatures for the constructors of nested datatypes need to quantify over the annotations of the inner datatypes. For example, we can introduce constructors for a list of boolean lists type:

$$\Sigma(\mathsf{lnil}) = \forall k_n, k_c, k'_n, k'_c.\ k'_n \to \mathsf{listlist}(k_n, k_c, k'_n, k'_c)$$
$$\text{and} \quad \Sigma(\mathsf{lcons}) = \forall k_n, k_c, k'_n, k'_c.\ \mathsf{boollist}(k_n, k_c), \mathsf{listlist}(k_n, k_c, k'_n, k'_c), k'_c \to \mathsf{listlist}(k_n, k_c, k'_n, k'_c).$$

We can now define the function to assign potential to typed values by summing the annotations over every reachable value:

$$\Upsilon : \mathsf{heap} \times \mathsf{val} \times T_a \to \mathbb{Q}^+,$$
$$\Upsilon(\sigma, *, 1) = \Upsilon(\sigma, \mathsf{true}, \mathsf{bool}) = \Upsilon(\sigma, \mathsf{false}, \mathsf{bool}) = 0$$
$$\Upsilon(\sigma, (v', v''), T' \otimes T'') = \Upsilon(\sigma, v', T') + \Upsilon(\sigma, v'', T''),$$
$$\Upsilon(\sigma, \mathsf{inl}(v), (T', k') + (T'', k'')) = k' + \Upsilon(\sigma, v, T'),$$
$$\Upsilon(\sigma, \mathsf{inr}(v), (T', k') + (T'', k'')) = k'' + \Upsilon(\sigma, v, T''),$$
$$\Upsilon(\sigma, \mathsf{null}, ty(\overline{k})) = k_i \quad \text{where} \quad c \in \mathsf{nullc} \quad \text{and} \quad \Sigma(c)[\overline{k}] = k_i \to ty(\overline{k}),$$
$$\Upsilon(\sigma, l, ty(\overline{k})) = \sum_{i=1}^{p} \Upsilon(\sigma \setminus l, v_i, T_i) + k_j, \quad \text{where} \quad \sigma(l) = (c, v_1, \ldots, v_p), \quad \text{and} \quad \Sigma(c)[\overline{k}] = T_1, \ldots, T_p, k_j \to ty(\overline{k}).$$

We can extend it to environments:

$$\Upsilon(\sigma, S, \Gamma) = \sum_{x \in \mathrm{dom}(\Gamma)} \Upsilon(\sigma, S(x), \Gamma(x)).$$

Thus the bound on the free memory required to evaluate an expression is the potential from the data, $\Upsilon(\sigma, S, \Gamma)$, plus the fixed amount from the typing judgement.

Continuing our list example, if x is a list of booleans then $\Upsilon(\sigma, S(x), \mathsf{boollist}(k_n, k_c))$ is k_c times the length of x plus k_n. Every list contains exactly one nil, so k_n could be safely set to zero because there is always an equivalent typing where k_n has been incorporated into the fixed amounts in the judgements. Hence we will use $\mathsf{boollist}(k)$ as a shorthand for $\mathsf{boollist}(0,k)$ in our examples.

The typing rules for expressions in the Hofmann-Jost system are given in Figures 2.5 and 2.6, and uses the same size function as the operational semantics. The typing judgements take the form

$$\Gamma, n \vdash_{\Sigma,F} e : T, n' \,|\, \Phi$$

where Γ is the typing context, n is the annotation for the fixed amount of potential before evaluation (in addition to that from the type annotations in Γ), n' is the corresponding annotation for potential after evaluation, T is the annotated type of e, Σ contains the function signatures, F is the set of function names defined in earlier blocks of mutually recursive definitions and Φ is the set of constraints on annotations that must hold for a valid typing. The pattern matching rules CASE and CASE$'$ also have the type being matched, $ty(\overline{k})$, in addition to the normal context:

$$\Gamma, n \mathbin{!} ty(\overline{k}) \vdash_{\Sigma,F} p \to e : T, n' \,|\, \Phi$$

This allows us to remove the matched variable from the context, but keep the type present so that we can use the type's annotations in the rules.

The additional rules in Figure 2.7 check that mutually recursive blocks of functions and entire programs are well typed, with functions conforming to their function signatures in Σ. Note that all of the constraint sets for the functions in a mutually recursive block are gathered together and put in all of the signatures. This is necessary for the FUNDEF rule to be sound. We say that a program P is *well-typed* if it satisfies $\vdash_{\Sigma,\emptyset} P$.

The 'leaf' rules are similar to those in a normal type system, with the exception of the function application rules. The key difference from a normal type system is the constraint given for the annotations. For example, the CONSTRUCT rule requires that

$$n \geq \mathrm{size}(c_i) + k_i + n',$$

meaning that we reduce the fixed amount of free memory by at least the size of the allocation and the increase in potential of the data structure compared to its arguments, k_i. The constraints maintain the invariant that the free memory is at least as large as the potential.

$$\frac{}{\Gamma, n \vdash_{\Sigma,F} * : 1, n' \,|\, \{n \geq n'\}} \text{ (UNIT)} \qquad \frac{c \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma, n \vdash_{\Sigma,F} c : \mathsf{bool}, n' \,|\, \{n \geq n'\}} \text{ (BOOL)}$$

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma, n \vdash_{\Sigma,F} x : \Gamma(x), n' \,|\, \{n \geq n'\}} \text{ (VAR)}$$

$$\frac{\begin{array}{c} f \in F \qquad \Sigma(f) = T'_1, \ldots, T'_p, k \to T', k' | \Phi' \qquad \rho(T'_i) = T_i \qquad \rho(T') = T \\ \Phi = \rho(\Phi') \cup \{n \geq \rho(k), n - \rho(k) + \rho(k') \geq n'\} \end{array}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash_{\Sigma,F} f(x_1, \ldots, x_p) : T, n' \,|\, \Phi} \text{ (FUN)}$$

$$\frac{\begin{array}{c} f \notin F \qquad \Sigma(f) = T_1, \ldots, T_p, k \to T, k' | \Phi' \\ \Phi = \{n \geq k, n - k + k' \geq n'\} \end{array}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash_{\Sigma,F} f(x_1, \ldots, x_p) : T, n' \,|\, \Phi} \text{ (FUNDEF)}$$

$$\frac{\Gamma_1, n \vdash_{\Sigma,F} e_1 : T_0, n_0 \,|\, \Phi_1 \qquad \Gamma_2, x : T_0, n_0 \vdash_{\Sigma,F} e_2 : T, n' \,|\, \Phi_2}{\Gamma_1, \Gamma_2, n \vdash_{\Sigma,F} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T, n' \,|\, \Phi_1 \cup \Phi_2} \text{ (LET)}$$

$$\frac{\Gamma, n \vdash_{\Sigma,F} e_t : T, n' \,|\, \Phi_1 \qquad \Gamma, n \vdash_{\Sigma,F} e_f : T, n' \,|\, \Phi_2}{\Gamma, x : \mathsf{bool}, n \vdash_{\Sigma,F} \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : T, n' \,|\, \Phi_1 \cup \Phi_2} \text{ (IF)}$$

$$\frac{}{\Gamma, x_1 : T_1, x_2 : T_2, n \vdash_{\Sigma,F} (x_1, x_2) : T_1 \otimes T_2, n' \,|\, \{n \geq n'\}} \text{ (PAIR)}$$

$$\frac{\Gamma, x_1 : T_1, x_2 : T_2, n \vdash_{\Sigma,F} e : T, n' \,|\, \Phi}{\Gamma, x : T_1 \otimes T_2, n \vdash_{\Sigma,F} \mathsf{match}\ x\ \mathsf{with}\ (x_1, x_2) \to e : T, n' \,|\, \Phi} \text{ (PAIRELIM)}$$

$$\frac{}{\Gamma, x : T_l, n \vdash_{\Sigma,F} \mathsf{inl}(x) : (T_l, k_l) + (T_r, k_r), n' \,|\, \{n \geq k_l + n'\}} \text{ (INL)}$$

$$\frac{}{\Gamma, x : T_r, n \vdash_{\Sigma,F} \mathsf{inr}(x) : (T_l, k_l) + (T_r, k_r), n' \,|\, \{n \geq k_r + n'\}} \text{ (INR)}$$

$$\frac{\begin{array}{c} \Gamma, x_l : T_l, n_l \vdash_{\Sigma,F} e_l : T, n' \,|\, \Phi_l \qquad \Gamma, x_r : T_r, n_r \vdash_{\Sigma,F} e_r : T, n' \,|\, \Phi_r \\ \Phi = \Phi_l \cup \Phi_r \cup \{n_l = n + k_l, n_r = n + k_r\} \end{array}}{\Gamma, x : (T_l, k_l) + (T_r, k_r), n \vdash_{\Sigma,F} \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r : T, n' \,|\, \Phi} \text{ (SUMELIM)}$$

$$\frac{\Gamma, a : T_1, b : T_2, n \vdash_{\Sigma,F} e : T', n' \,|\, \Phi \qquad T = T_1 \oplus T_2 \,|\, \Phi'}{\Gamma, x : T, n \vdash_{\Sigma,F} e[x/a, x/b] : T', n' \,|\, \Phi \cup \Phi'} \text{ (SHARE)}$$

Figure 2.5: Typing rules for expressions in the Hofmann-Jost analysis

$$\frac{\begin{array}{c}\Sigma(c_i)[\bar{k}] = T_1,\ldots,T_p,k_i \to ty(\bar{k}) \\ \Phi = \{n \geq \text{size}(c_i) + k_i + n'\}\end{array}}{\Gamma, x_1 : T_1,\ldots,x_p : T_p, n \vdash_{\Sigma,F} c_i(x_1,\ldots,x_p) : ty(\bar{k}), n' \,|\, \Phi} \text{ (CONSTRUCT)}$$

$$\frac{\text{for all } i,\ 1 \leq i \leq m, \quad \Gamma, n \dashv ty(\bar{k}) \vdash_{\Sigma,F} p_i \to e_i : T', n' \,|\, \Phi_i}{\Gamma, x : ty(\bar{k}), n \vdash_{\Sigma,F} \text{match } x \text{ with } p_1 \to e_1 \dashv \cdots \dashv p_m \to e_m : T', n' \,|\, \bigcup_i \Phi_i} \text{ (MATCH)}$$

$$\frac{\begin{array}{c}\Sigma(c_i)[\bar{k}] = T_1,\ldots,T_p,k_i \to ty(\bar{k}) \\ \Gamma, x_1 : T_1,\ldots,x_p : T_p, n_i \vdash_{\Sigma,F} e : T', n' \,|\, \Phi \\ \Phi' = \{n_i = n + k_i + \text{size}(c_i)\}\end{array}}{\Gamma, n \dashv ty(\bar{k}) \vdash_{\Sigma,F} c_i(x_1,\ldots,x_p) \to e : T', n' \,|\, \Phi \cup \Phi'} \text{ (CASE)}$$

$$\frac{\begin{array}{c}\Sigma(c_i)[\bar{k}] = T_1,\ldots,T_p,k_i \to ty(\bar{k}) \\ \Gamma, x_1 : T_1,\ldots,x_p : T_p, n_i \vdash_{\Sigma,F} e : T', n' \,|\, \Phi \\ \Phi' = \{n_i = n + k_i\}\end{array}}{\Gamma, n \dashv ty(\bar{k}) \vdash_{\Sigma,F} c_i(x_1,\ldots,x_p)' \to e : T', n' \,|\, \Phi \cup \Phi'} \text{ (CASE}') \text{ (CASE')}$$

Figure 2.6: Typing rules for expressions in the Hofmann-Jost analysis (continued)

$$\frac{\Sigma(f) = T_1,\ldots,T_p,k \to T,k' | \Phi \qquad x_1 : T_1,\ldots,x_p : T_p,k \vdash_{\Sigma,F} e_f : T,k' \,|\, \Phi'}{\vdash_{\Sigma,F} f(x_1,\ldots,x_p) = e_f \Rightarrow \{f\}, \Phi'}$$

$$\frac{\vdash_{\Sigma,F} D \Rightarrow F', \Phi' \qquad \vdash_{\Sigma,F} B \Rightarrow F'', \Phi''}{\vdash_{\Sigma,F} D \text{ and } B \Rightarrow F' \cup F'', \Phi' \cup \Phi''}$$

$$\frac{\begin{array}{c}\vdash_{\Sigma,F} B \Rightarrow F', \Phi' \\ \forall f \in F'. \quad \Sigma(f) = \ldots | \Phi'\end{array}}{\vdash_{\Sigma,F} \text{let } B} \qquad \frac{\begin{array}{c}\vdash_{\Sigma,F} B \Rightarrow F', \Phi' \qquad \vdash_{\Sigma,F \cup F'} P \\ \forall f \in F'. \quad \Sigma(f) = \ldots | \Phi'\end{array}}{\vdash_{\Sigma,F} \text{let } B\ P}$$

Figure 2.7: Typing rules for function signatures

$$\frac{}{1 = 1 \oplus 1 \,|\, \emptyset} \qquad \frac{}{\text{bool} = \text{bool} \oplus \text{bool} \,|\, \emptyset}$$

$$\frac{T = T_1 \oplus T_2 \,|\, \Phi \qquad T' = T_1' \oplus T_2' \,|\, \Phi'}{T \otimes T' = (T_1 \otimes T_1') \oplus (T_2 \otimes T_2') \,|\, \Phi \cup \Phi'}$$

$$\frac{T = T_1 \oplus T_2 \,|\, \Phi \qquad T' = T_1' \oplus T_2' \,|\, \Phi' \qquad \Phi'' = \{k = k_1 + k_2, k' = k_1' + k_2'\}}{(T,k) + (T',k') = (T_1,k_1) + (T_1',k_1') \oplus (T_2,k_2) + (T_2',k_2') \,|\, \Phi \cup \Phi' \cup \Phi''}$$

$$\frac{}{ty(\bar{k}) = ty(\overline{k_1}) \oplus ty(\overline{k_2}) \mid \{k_i = k_{1,i} + k_{2,i} : \forall i\}}$$

Figure 2.8: Rules for splitting annotations

The inductive rules must provide constraints linking the fixed amounts used in typing the expression to those in the judgements for its subexpressions. As with the leaf rules, these constraints must reflect changes in allocation and potential. In particular, the CASE rule has a complementary constraint to CONSTRUCT,

$$n_i = n + k_i + \text{size}(c_i),$$

with the difference that n_i is used in the judgement for the subexpression, rather than as the fixed amount *after* the evaluation of the whole match expression. The set of constraints must contain the union of those from each subexpression in addition to these 'local' constraints.

The typing context is treated linearly; for example the MATCH rule does not provide the subexpressions with the variable being matched, x. This is prevent the duplication of the list's potential, which would lead to an underestimate of the memory requirements. Instead, we have an explicit contraction rule, SHARE, which divides the potential between uses of a variable by dividing up the annotation. The auxiliary rules in Figure 2.8 define this division, which ensures that the types' annotations sum pairwise to the combined type. For example, the judgement

$$\mathsf{boollist}(k) = \mathsf{boollist}(k_1) \oplus \mathsf{boollist}(k_2) \,|\, \{k = k_1 + k_2\}$$

allows $\mathsf{boollist}(3) = \mathsf{boollist}(2) \oplus \mathsf{boollist}(1)$, splitting three units per element between two uses of the list. The rule can also be used to reduce an annotation so that two types match, because weakening of the typing context is admissible (which can be seen directly from the typing rules).

There are two rules for typing function applications. The FUN rule allows the resource polymorphism discussed above, which is embodied in a substitution ρ on constraint variables (extended to types and constraint sets). For inference we choose fresh names for every constraint variable that does not appear in the types $T'_1, \ldots, T'_p, T'$. Those appearing in the types are fixed by the side conditions $\rho(T'_i) = T_i$ and $\rho(T') = T$.

Resource polymorphism is restricted to applications of previously defined functions by checking that the function's name appears in the set F in the judgements. This is necessary during inference because the full set of constraints for the function being examined will not be known when typing a recursive call. Hence, the simpler FUNDEF rule is used to type such applications. This rule does not depend on the constraints in the function signature, and so can be used while inferring them.

2.2.3 Soundness

The soundness result of this analysis with respect to the operational semantics can now be given. The intuition is that any well typed expression can be executed with the amount of free memory predicted by the annotations ($n + \Upsilon(\sigma, S, \Gamma)$), and the annotation on the result conservatively predicts the amount of free memory afterwards. Moreover, execution will not consume any extra free memory (q) that may be available.

Theorem 2.4. *Suppose that* $\text{stack}(f) = 0$ *for all* f. *If an expression* e *in a function* f *in a well-typed program has a typing*

$$\Gamma, n \vdash_{\Sigma,F} e : T, n' \,|\, \Phi$$

with an assignment of nonnegative rationals to constraint variables which satisfies the constraints in f*'s signature, and an evaluation* $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ *which satisfies the benign sharing conditions, and* $\Upsilon(\sigma, S, \Gamma)$ *is defined, then for any* $q \in \mathbb{Q}^+$ *and* $m \in \mathbb{N}$ *such that*

$$m \geq n + \Upsilon(\sigma, S, \Gamma) + q$$

we have $m, S, \sigma \vdash e \rightsquigarrow v, \sigma', m'$ *where* $\Upsilon(\sigma', v, T)$ *is defined and*

$$m' \geq n' + \Upsilon(\sigma', v, T) + q.$$

A direct proof would be very similar to that in (Hofmann and Jost, 2003), so we merely note that it can also be viewed as a consequence of the soundness theorem in Chapter 5. A soundness result for non-terminating programs also holds for this system in the same way.

The proof essentially shows that any changes in the free memory during evaluation are conservatively approximated by changes in the potential. Aside from the addition of resource polymorphism and algebraic datatypes there is also a small technical difference in our presentation with respect to (Hofmann and Jost, 2003). We simplify the theorem and soundness proof by removing the judgement that the typing context, value environment and store are consistent. Instead, it is sufficient to note that $\Upsilon(\sigma, S, \Gamma)$ is defined only when the variables in Γ have values in S and σ that are of the correct type.

The soundness theorem leads to a bound on the evaluation of the initial function:

Corollary 2.5. *Suppose a well typed program has an initial function* f*, arguments for* f *are given as values* $v_1, \ldots, v_p$ *with an initial store* σ*, and* $\text{stack}(f) = 0$ *for all* f. *If*

$$\Sigma(f) = T_1, \ldots, T_p, k \to T', k' \,|\, \Phi$$

then any execution of $f(v_1,\ldots,v_p)$ *will require at most*

$$\Upsilon(\sigma,[x_1\mapsto v_1,\ldots,x_p\mapsto v_p],(x_1:T_1,\ldots,x_p:T_p))+k$$

units of memory, for any assignment of nonnegative rationals to constraint variables which satisfies Φ.

2.2.4 Examples

Now we can revisit the `notlist` and `id` examples to present typings for them. To make the type derivations more manageable we omit the weakening of annotations using SHARE, which is not required for these examples.

Example 2.6. Recall the `notlist` function defined in Example 2.1 on page 6. A typing for `notlist` is given in Figure 2.9, where the n_i constraint variables represent fixed amounts of potential, and the k_i constraint variables for the type annotations determine the potential of the data structures. We continue using $\mathsf{boollist}(k)$ as a shorthand for $\mathsf{boollist}(0,k)$. Note that the function application must be typed with the simpler FUN-DEF rule because it is recursive.

The `notlist` function allocates a new list without destroying the original one, so a linear amount of space is required. More precisely, when evaluating `notlist l` we will require

$$|\mathtt{l}|\times \mathrm{size}(\mathtt{cons})$$

units of heap memory. Thus we expect the bound to be represented by assigning $\mathrm{size}(\mathtt{cons})$ to k_1, the annotation on `l`'s type, so that the potential of `l` is large enough to allocate the result.

The sample solution for the constraints confirms this, and we can trace the requirements through the typing derivation. The CONSTRUCT rule for `cons` requires that n_5 is large enough to account for the allocation (Φ_2). The constraints on other parts of the expression thread this requirement back through n_4 and n_3, and so the amount to be allocated can be provided in one of three places: the fixed amount of free memory required to invoke the function (n_1), memory freed by the recursive function call (the $-n_1+n_2$ in Φ_1) or the drop in potential on matching the list (k_1).

The first two sources are ruled out by the recursive function call. If we try to raise n_1 to account for the allocation then we are required to find the same amount again for the recursive call. Similarly, if we assume that the recursive call finds some free

$$\mathcal{D}_1 = \dfrac{\dfrac{}{\cdot, n_3 \vdash_{\Sigma,\emptyset} \texttt{false} : \mathsf{bool}, n_4 \,|\, \{n_3 \geq n_4\}}\ \textsc{Bool} \quad \dfrac{}{\cdot, n_3 \vdash_{\Sigma,\emptyset} \texttt{true} : \mathsf{bool}, n_4 \,|\, \{n_3 \geq n_4\}}\ \textsc{Bool}}{\texttt{h} : \mathsf{bool}, n_3 \vdash_{\Sigma,\emptyset} \texttt{if}\ \cdots : \mathsf{bool}, n_4 \,|\, \{n_3 \geq n_4\}}\ \textsc{If}$$

$$\mathcal{D}_2 = \dfrac{\dfrac{}{\texttt{t} : \mathsf{boollist}(k_1), n_4 \vdash_{\Sigma,\emptyset} \texttt{notlist t} : \mathsf{boollist}(k_2), n_5 \,|\, \Phi_1}\ \textsc{FunDef} \quad \dfrac{}{\texttt{hh} : \mathsf{bool}, \texttt{tt} : \mathsf{boollist}(k_2), n_5 \vdash_{\Sigma,\emptyset} \texttt{cons} \cdots : \mathsf{boollist}(k_2), n_2 \,|\, \Phi_2}\ \textsc{Construct}}{\texttt{t} : \mathsf{boollist}(k_1), \texttt{hh} : \mathsf{bool}, n_4 \vdash_{\Sigma,\emptyset} \texttt{let tt} \cdots : \mathsf{boollist}(k_2), n_2 \,|\, \Phi_1 \cup \Phi_2}\ \textsc{Let}$$

$$\dfrac{\dfrac{\dfrac{}{\cdot, n_1 \vdash_{\Sigma,\emptyset} \texttt{nil} : \mathsf{boollist}(k_2), n_2 \,|\, \{n_1 \geq n_2\}}\ \textsc{Construct}}{\cdot, n_1 \triangleleft \mathsf{boollist}(k_1) \vdash_{\Sigma,\emptyset} \texttt{nil}' \rightarrow \texttt{nil} : \mathsf{boollist}(k_2), n_2 \,|\, \{n_1 \geq n_2\}}\ \textsc{Case}' \quad \dfrac{\dfrac{\overset{\mathcal{D}_1}{\cdots} \quad \overset{\mathcal{D}_2}{\cdots}}{\texttt{h} : \mathsf{bool}, \texttt{t} : \mathsf{boollist}(k_1), n_3 \vdash_{\Sigma,\emptyset} \texttt{let hh} \cdots : \mathsf{boollist}(k_2), n_2 \,|\, \Phi_3}\ \textsc{Let}}{\cdot, n_1 \triangleleft \mathsf{boollist}(k_1) \vdash_{\Sigma,\emptyset} \texttt{cons}(\texttt{h}, \texttt{t})' \rightarrow \cdots : \mathsf{boollist}(k_2), n_2 \,|\, \Phi_3 \cup \Phi_4}\ \textsc{Case}'}{\texttt{l} : \mathsf{boollist}(k_1), n_1 \vdash_{\Sigma,\emptyset} \texttt{match} \cdots : \mathsf{boollist}(k_2), n_2 \,|\, \Phi}\ \textsc{Match}$$

$$\dfrac{\texttt{l} : \mathsf{boollist}(k_1), n_1 \vdash_{\Sigma,\emptyset} \texttt{match} \cdots : \mathsf{boollist}(k_2), n_2 \,|\, \Phi}{\vdash_{\Sigma,\emptyset} \texttt{notlist l} = \ldots \Rightarrow \{\texttt{notlist}\}}$$

$$\vdash_{\Sigma,\emptyset} \texttt{let notlist l} = \ldots$$

$$\Phi_1 = \{n_4 \geq n_1, n_4 - n_1 + n_2 \geq n_5\} \qquad \Phi_3 = \{n_3 \geq n_4\} \cup \Phi_1 \cup \Phi_2$$

$$\Phi_2 = \{n_5 \geq \mathrm{size}(\texttt{cons}) + k_2 + n_2\} \qquad \Phi_4 = \{n_3 = n_1 + k_1\}$$

$$\Phi = \Phi_3 \cup \{n_1 \geq n_2\} \cup \Phi_4$$

$$\Sigma = \begin{bmatrix} \texttt{nil} \mapsto \forall k.\mathsf{boollist}(k) \\ \texttt{cons} \mapsto \forall k.\mathsf{bool}, \mathsf{boollist}(k), k \rightarrow \mathsf{boollist}(k) \\ \texttt{notlist} \mapsto \mathsf{boollist}(k_1), n_1 \rightarrow \mathsf{boollist}(k_2), n_2 \,|\, \Phi \end{bmatrix}$$

Sample solution for Φ : $n_1 = n_2 = 0,\ n_3 = n_4 = n_5 = k_1 = \mathrm{size}(\texttt{cons}),\ k_2 = 0.$

Figure 2.9: `notlist` typing

memory $-n_1+n_2$, then we need to free the same amount again by the end of the caller. The only possibility left is to assign the cost to k_1, as expected.

Example 2.7. Recall the `id` function:

```
let id l = let notl = notlist l in notlist notl
```

Figure 2.10 gives a typing for `id`. In this function we use the resource polymorphic FUN rule for the two uses of `notlist`. This allows the two uses to be typed in the two different ways discussed in Section 2.2.1, which can be seen in the sample solution.

If `notlist` is modified to deallocate the supplied list then the deallocation can satisfy the memory requirements for the new list. This is realised by changing the typing derivation for `notlist` (Figure 2.9) to use CASE rather than CASE′, which changes the constraint to include the extra free memory:

$$\Phi'_4 = \{n_3 = n_1 + k_1 + \text{size}(\texttt{cons})\}.$$

Hence k_1 can be zero, yielding the signature

$$\texttt{notlist} : \mathsf{boollist}(0), 0 \to \mathsf{boollist}(0), 0$$

showing that no extra memory is required.

This signature for `notlist` can then be used for both function applications when typing `id`. As a result, the same signature can be derived for `id`, indicating that it can be evaluated in-place.

The next example illustrates why we allow rational solutions rather than restricting ourselves to the integers.

Example 2.8. Consider the following function:

```
let evens l = match  l with nil' -> nil | cons(h1,t1)' ->
              match t1 with nil' -> nil | cons(h2,t2)' ->
              let t' = evens t2 in cons(h2,t')
```

This takes a list and creates a new list with every second element of the original. The analysis gives this function the signature (after constraint solving) of

$$\texttt{evens} : \mathsf{boollist}(\tfrac{1}{2}), 0 \to \mathsf{boollist}(0), 0$$

because we allocate a list *half* as long as the argument.

$$\mathcal{D}_3 = \frac{}{\texttt{l} : \mathsf{boollist}(l_1), m_1 \vdash_{\Sigma,F} \texttt{notlist l} : \mathsf{boollist}(l_3), m_3 \,|\, \rho_1(\Phi) \cup \{m_1 \geq \rho_1(n_1), m_1 - \rho_1(n_1) + \rho_1(n_2) \geq m_3\}}\ \textsc{Fun}$$

$$\mathcal{D}_4 = \frac{}{\texttt{notl} : \mathsf{boollist}(l_3), m_3 \vdash_{\Sigma,F} \texttt{notlist notl} : \mathsf{boollist}(l_2), m_2 \,|\, \rho_2(\Phi) \cup \{m_3 \geq \rho_2(n_1), m_3 - \rho_2(n_1) + \rho_2(n_2) \geq m_2\}}\ \textsc{Fun}$$

$$\frac{\dfrac{\text{(as in Figure 2.9)}}{\vdash_{\Sigma,\emptyset} \texttt{notlist l} = \ldots \Rightarrow \{\texttt{notlist}\}} \qquad \dfrac{\dfrac{\dfrac{\mathcal{D}_3 \qquad \mathcal{D}_4}{\texttt{l} : \mathsf{boollist}(l_1), m_1 \vdash_{\Sigma,F} \texttt{let notl} \cdots : \mathsf{boollist}(l_2), m_2 \,|\, \Phi'}\ \textsc{Let}}{\vdash_{\Sigma,F} \texttt{id l} = \ldots \Rightarrow \{\texttt{id}\}}}{\vdash_{\Sigma,F} \texttt{let id l} = \ldots \Rightarrow \{\texttt{id}\}}}{\vdash_{\Sigma,\emptyset} \texttt{let notlist} = \ldots \texttt{ let id} = \ldots}$$

$$\Phi' = \rho_1(\Phi) \cup \rho_2(\Phi) \cup \left\{ \begin{array}{l} m_1 \geq \rho_1(n_1), m_1 - \rho_1(n_1) + \rho_1(n_2) \geq m_3, \\ m_3 \geq \rho_2(n_1), m_3 - \rho_2(n_1) + \rho_2(n_2) \geq m_2 \end{array} \right\}$$

$$\Sigma = \left[\begin{array}{rl} \texttt{nil} & \mapsto \forall k.\mathsf{boollist}(k) \\ \texttt{cons} & \mapsto \forall k.\mathsf{bool}, \mathsf{boollist}(k), k \rightarrow \mathsf{boollist}(k) \\ \texttt{notlist} & \mapsto \mathsf{boollist}(k_1), n_1 \rightarrow \mathsf{boollist}(k_2), n_2 \,|\, \Phi \\ \texttt{id} & \mapsto \mathsf{boollist}(l_1), m_1 \rightarrow \mathsf{boollist}(l_2), m_2 \,|\, \Phi' \end{array} \right]$$

Sample solution for `id`: $m_1 = m_2 = m_3 = 0,\ l_2 = \rho_2(k_2) = 0, \rho_1(n_1) = \rho_1(n_2) = \rho_2(n_1) = \rho_2(n_2) = 0,$
$l_1 = \rho_1(k_1) = 2 \times \text{size}(\texttt{cons}),\ l_3 = \rho_1(k_2) = \rho_2(k_1) = \text{size}(\texttt{cons}).$

Figure 2.10: `id` typing

A more demanding example is a functional implementation of heap sort. Most imperative implementations work in-place on the array of data to be sorted, using the array indices to keep track of the heap structure. In contrast, a purely functional version may explicitly build a heap data structure, see (Paulson, 1996, Section 4.16).

The key properties of the imperative version which allow the in-place update are using the indices to provide the heap structure (so no extra structure needs to be allocated) and the lack of any need to look at an old copy of the data structure. We do not need to refer to old versions of the list and heap in the functional version either; so list elements can be deallocated as they are put into the heap, and vice versa on extracting the sorted elements.

Example 2.9. See Appendix A for the functional version of heap sort, written in LFD augmented by support for integers in addition to booleans. The analysis takes advantage of the deallocation to infer a signature (after constraint solving) of

$$\texttt{sort} : \mathsf{intlist}(k), 0 \to \mathsf{intlist}(k), 0,$$
$$k = \mathrm{size}(\texttt{node}) - \mathrm{size}(\texttt{cons}),$$

meaning that we only need enough extra memory to turn the list elements into tree nodes, and that the extra memory is free again after evaluation. If we make list elements as large as tree nodes then $k = 0$ and we regain the in-place behaviour of the imperative version.

More detail may be found in Appendix A.

2.2.5 Inference and Complexity

Given a program, it is easy to infer an unannotated typing using standard unification techniques. We then wish to infer an annotated typing, including a set of constraints for each function, and finally a solution for the constraint set of the initial function.

We first add annotations to the types in the form of constraint variables, and use the typing rules to build the constraint set for each function. There are only three rules that are not syntax-directed: FUN, FUNDEF and SHARE. The choice between the two application rules is broken by consulting the set of previously defined functions, F, as described in Section 2.2.2.

We need to use the SHARE rule in two circumstances. The first use is for contraction, where a variable will be required multiple times during evaluation. For instance, if

a variable appears twice in the arguments of a function (`f(x,x)`), or in different subexpressions of a let (`let y=x in f(x,y)`) then we need to use SHARE for contraction, but not in the different branches of an if because the context is not split between them. The SHARE rule divides the potential between the different uses of the variable. We can place the SHARE rule at any point in the derivation between the binding of the variable and the point at which it is required multiple times.

The second use of SHARE is to weaken type annotations. Rules involving multiple subexpressions require their types to match, so we may wish to 'leak' some potential. For example, in

```
if b then l else notlist l
```

one branch of computation allocates an extra list. If `l` has type $\mathsf{boollist}(1)$ then `notlist l` will have the type $\mathsf{boollist}(0)$. By adding a use of the SHARE rule we can weaken the left hand `l`'s annotation to match:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{l}_1 : \mathsf{boollist}(0), \mathtt{l}_2 : \mathsf{boollist}(1), 0 \vdash \mathtt{l}_1 : \mathsf{boollist}(0), 0}\,\textsc{Var}}{\mathtt{l} : \mathsf{boollist}(1), 0 \vdash \mathtt{l} : \mathsf{boollist}(0), 0}\,\textsc{Share} \qquad \dots}{\mathtt{b} : \mathsf{bool}, \mathtt{l} : \mathsf{boollist}(1), 0 \vdash \mathtt{if}\ \cdots : \mathsf{boollist}(0), 0}\,\textsc{If}$$

(Using solutions rather than constraints for brevity.) In general, it is sufficient to weaken at the leaf rules which may yield annotated types: VAR, FUN, FUNDEF, PAIR, INL, INR and CONSTRUCT.

Once we have the annotated typing we can solve the constraint set for the initial function using standard linear programming techniques. The remaining task is to choose an objective function to minimise the bound found. Jost's implementation takes the annotations from the function signature and applies a simple scoring system (Jost, 2004b). In particular, data structure annotations are more 'expensive' than fixed amounts. With a little care, negative scores can be placed on the annotations on the right hand side of the signature to maximise the bound on the memory free after evaluation[4].

An alternative approach is to consider the overall bound from Corollary 2.5,

$$\Upsilon(\sigma, [x_1 \mapsto v_1, \dots, x_p \mapsto v_p], (x_1 : T_1, \dots, x_p : T_p)) + k.$$

[4]The care is required because an unwise choice of negative scores can yield unbounded solutions. For example, in

```
let fnil x = nil : *,0 → T list(k),0
```

k is unconstrained. Thus a negative score on k will give an unbounded objective function.

The potential function Υ essentially multiplies each annotation k_i by the size of the data structures involved. For example, a nested list x of type $\mathsf{listlist}(k_1, k_2)$ (the type of list of lists of booleans introduced on page 19, with 0 for the nil annotations) has potential $k_1 \times |x| + \sum_{y \in x} k_2 \times |y|$. Thus if we assign to every annotation a coefficient that is the expected size of the corresponding data we obtain a bound optimised for the expected case.

Once we have a bound on the free memory required we can add an extra constraint to the linear program to fix that bound, then solve the new linear program with an objective function which maximises the bound on the free memory after evaluation in the same way.

Note that there is no 'best' objective function because it is not always possible to express the optimal bound in the above form (an affine function on data structure sizes). To see this, consider the following example:

Example 2.10. The family of functions

```
let f l = let l0 = nil in
          match l with nil' -> l0 | cons(_,t1)' ->
          let l1 = cons(true,l0) in
          match t1 with nil' -> l1 | cons(_,t2)' ->
          ...
          let ln = cons(true,l⟨n − 1⟩) in
          ln
```

constructs a list of length $\min\{|\mathtt{l}|, n\}$. However, the Hofmann-Jost system can infer only one of $|\mathtt{l}|$ or n depending upon the choice of objective function.

Jost implemented the original system (without resource polymorphism, essentially using the FUNDEF rule for all function applications) extended with user-defined algebraic datatypes (Jost, 2004b). It was used in the Mobile Resource Guarantees project to produce resource bounds for an intermediate language of the Camelot compiler.

A version with resource polymorphism was produced by the author as part of the work described in Chapter 4. Also, Jost's implementation of his later system with higher-order functions, ARTHUR, contains a similar mechanism (Jost, 2004a).

We now turn to the complexity of the inference. If resource polymorphism is ignored (by using FUNDEF for all applications) then all of the typing rules bar SHARE are syntax directed and contribute a constant number of constraints per use. The SHARE

rule requires as many constraints as there are annotations in the type of the variable in question. Thus so long as SHARE is only used for 'real' contraction and weakening then the number of constraints generated is $O(\text{program size} \times \text{number of constructors})$.

The resulting linear program can be solved in polynomial time, and so the entire type inference can be performed in polynomial time with respect to the size of the original program.

Adding resource polymorphism, however, allows an exponential growth in the number of constraints with the size of the program:

Example 2.11. Consider the family of programs of the form:

```
let id1 x = x
let id2 x = let y = id1 x in id1 y
let id3 x = let y = id2 x in id2 y
let id4 x = let y = id3 x in id3 y
let id5 x = let y = id4 x in id4 y
...
```

Each function has a signature of the form

$$\texttt{id}i : T, n_i \to T, n_i' \,|\, \Phi_i,$$

with $\Phi_1 = \{n_1 \geq n_1'\}$. Each subsequent constraint set Φ_{i+1} contains two copies of the previous one with the variables renamed, say $\Phi_{i,1}$ and $\Phi_{i,2}$. Then by FUN and LET,

$$\Phi_{i+1} = \Phi_{i,1} \cup \Phi_{i,2} \cup \{n_{i+1} \geq n_{i,1}, n_{i,1}' \geq n_{i,2}, n_{i,2}' \geq n_{i+1}\},$$

consisting of the two copies of the previous set, and three new constraints linking the annotations for the fixed amounts of free memory at the start, middle and end of the function's evaluation.

Thus each Φ_i is more than double the size of the previous one.

It is not clear if the linear programs produced for functions could be simplified to avoid this exponential complexity. We leave this question to future work.

Chapter 3

Using a CPS transformation to bound stack requirements

We noted in the previous chapter that the LFD language used in this thesis can be viewed as a compiler's intermediate language. In particular, the Camelot compiler used in the Mobile Resource Guarantees project performed its monomorphisation and let-normalisation stages to produce LFD code for Hofmann and Jost's analysis (Jost, 2004b). This removed the burden of handling polymorphism and evaluation order from the analysis. In this chapter we consider using a further compiler stage to remove the burden of handling stack space.

Some compilers use an intermediate language in *Continuation Passing Style* (CPS). In programs of this form every function application is a tail call which takes a continuation function as an argument to represent the remainder of the program. CPS has similar advantages to let-normal form: evaluation order is explicit and intermediate values are named. Another benefit is that the control flow is closer to the final machine code because the continuations correspond to the link register and stack typically found in compiled code.

This exposes some of the stack manipulation, and also gives the implementor the choice of storing the frames on the heap instead. (For instance, (Appel, 1992, §10.8) discusses a tradeoff where heap allocation of frames makes the use of first class continuations cheaper.) Thus we are interested in using a CPS transformation to make the frames into explicit heap allocated structures and then inferring bounds on their size using Hofmann and Jost's heap analysis without alteration.

First we will define the transformation itself and show that the original analysis can be used to obtain total bounds on the resulting programs. Then we establish the

correctness of the transformation and consider using the analysis of the transformed program to give bounds on the original program. Finally we discuss some examples and difficulties which arise when using the CPS transformation for analysis, and motivate the direct approach in the following chapters.

3.1 The CPS transformation

CPS transformations have a long history, both in compilers such as RABBIT (Steele, 1978) and SML/NJ (Appel, 1992), and in theoretical uses (for example (Plotkin, 1975)). (Reynolds, 1993) provides a review of the early work using continuations and CPS. Ironically, recent work tends toward using A-normal form in intermediate languages rather than CPS (Flanagan et al., 2004) and the let-normal form of LFD reflects this, although (Kennedy, 2007) makes a strong case for CPS using second class continuations. Nonetheless, CPS is still of interest to us as a mechanism to bound stack usage.

Our CPS transformation is a little unusual as a result of choosing LFD as the source and target language. LFD is a first-order language which prevents us from using real continuations. Instead we use a defunctionalized CPS transform, where we construct explicit closures for our continuations. Also, LFD has a simple type system which forces us to add some extra complexity to maintain typability.

The main part of our transformation acts on expressions with unannotated type information. It is presented in Figure 3.1 as a set of syntax-directed rules with judgements of the form

$$\Gamma \vdash_{\Sigma} e : T, x, e' \mapsto e'', (F, S, R)$$

where Γ is the typing context, Σ the source function signatures, e is the expression (of type T) to be transformed, e' is the continuation expression (which expects the result of e to be bound to x), e'' is the transformed expression and the tuple (F, S, R) represents information about the new continuation functions. The continuation expression is the previously transformed expression representing 'the rest of the program.' It is not a 'proper' continuation because it may become part of the transformed expression e'' rather than a continuation function. T_{final} is the result type of the initial function and s and s' are two fresh variable names used consistently throughout the transformation. We use $\langle' \rangle$ in the C-MATCH rule to denote an optional $'$.

The new continuation function information consists of a tuple (F, S, R) where the first component F is the set of new functions, the second S is a map from new constructors to the constructor's type signature, and the third R is a map from types to the

$$\frac{}{\Gamma \vdash_\Sigma * : 1, x, e' \mapsto \mathsf{let}\ x = *\ \mathsf{in}\ e', \emptyset} \quad \text{(C-UNIT)}$$

$$\frac{}{\Gamma \vdash_\Sigma c : \mathsf{bool}, x, e' \mapsto \mathsf{let}\ x = c\ \mathsf{in}\ e', \emptyset} \quad \text{(C-BOOL)}$$

$$\frac{}{\Gamma \vdash_\Sigma x_0 : T, x, e' \mapsto \mathsf{let}\ x = x_0\ \mathsf{in}\ e', \emptyset} \quad \text{(C-VAR)}$$

$$\frac{}{\Gamma \vdash_\Sigma (x_1, x_2) : T, x, e' \mapsto \mathsf{let}\ x = (x_1, x_2)\ \mathsf{in}\ e', \emptyset} \quad \text{(C-PAIR)}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{inl}(x_0) : T, x, e' \mapsto \mathsf{let}\ x = \mathsf{inl}(x_0)\ \mathsf{in}\ e', \emptyset} \quad \text{(C-INL)}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{inr}(x_0) : T, x, e' \mapsto \mathsf{let}\ x = \mathsf{inr}(x_0)\ \mathsf{in}\ e', \emptyset} \quad \text{(C-INR)}$$

$$\frac{}{\Gamma \vdash_\Sigma c(x_1, \ldots, x_p) : T, x, e' \mapsto \mathsf{let}\ x = c(x_1, \ldots, x_p)\ \mathsf{in}\ e', \emptyset} \quad \text{(C-CONSTRUCT)}$$

$$\frac{\begin{array}{c}\text{fresh } C \qquad \{z_1, \ldots, z_n\} = \mathrm{FV}(e') \backslash \{x, s\} \\ K = (\mathsf{cont}C(x, z_1, \ldots, z_n, s) = e', \left[\begin{array}{c} C \mapsto (\Gamma(z_1), \ldots, \Gamma(z_n), \mathsf{stack} \to \mathsf{stack}), \\ \mathsf{cont}C \mapsto (T, \Gamma(z_1), \ldots, \Gamma(z_n), \mathsf{stack} \to T_{\text{final}}) \end{array}\right], [T \mapsto \{C\}])\end{array}}{\Gamma \vdash_\Sigma f(x_1, \ldots, x_p) : T, x, e' \mapsto \mathsf{let}\ s' = C(z_1, \ldots, z_n, s)\ \mathsf{in}\ f'(x_1, \ldots, x_p, s'), K} \quad \text{(C-FUN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e_1 : T, x_1, e_2' \mapsto e_1', K_1 \\ \Gamma, x_1 : T \vdash_\Sigma e_2 : T', x, e' \mapsto e_2', K_2\end{array}}{\Gamma \vdash_\Sigma \mathsf{let}\ x_1 : T = e_1\ \mathsf{in}\ e_2 : T', x, e' \mapsto e_1', K_1 \uplus K_2} \quad \text{(C-LET)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e_1 : T, x, e' \mapsto e_1', K_1 \\ \Gamma \vdash_\Sigma e_2 : T, x, e' \mapsto e_2', K_2\end{array}}{\Gamma \vdash_\Sigma \mathsf{if}\ x_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T, x, e' \mapsto \mathsf{if}\ x_0\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2', K_1 \uplus K_2} \quad \text{(C-IF)}$$

$$\frac{\Gamma(x_0) = T_1 \otimes T_2 \qquad \Gamma, x_1 : T_1, x_2 : T_2 \vdash_\Sigma e_1 : T, x, e' \mapsto e_1', K_1}{\Gamma \vdash_\Sigma \mathsf{match}\ x_0\ \mathsf{with}\ (x_1, x_2) \to e_1 : T, x, e' \mapsto \mathsf{match}\ x_0\ \mathsf{with}\ (x_1, x_2) \to e_1', K_1} \quad \text{(C-MATCHPAIR)}$$

$$\frac{\begin{array}{c}\Gamma(x_0) = T_1 + T_2 \\ \Gamma, x_1 : T_1 \vdash_\Sigma e_1 : T, x, e' \mapsto e_1', K_1 \\ \Gamma, x_2 : T_2 \vdash_\Sigma e_2 : T, x, e' \mapsto e_2', K_2\end{array}}{\begin{array}{l}\Gamma \vdash_\Sigma \mathsf{match}\ x_0\ \mathsf{with}\ \mathsf{inl}(x_1) \to e_1 \mid \mathsf{inr}(x_2) \to e_2 : T, x, e' \\ \quad \mapsto \mathsf{match}\ x_0\ \mathsf{with}\ \mathsf{inl}(x_1) \to e_1' \mid \mathsf{inr}(x_2) \to e_2', K_1 \uplus K_2\end{array}} \quad \text{(C-MATCHSUM)}$$

$$\frac{\text{for all } i,\ 1 \le i \le m. \quad \left\{\begin{array}{l}\Sigma(c_i) = T_{i,1}, \ldots, T_{i,p_i} \to ty \\ \Gamma, x_{i,1} : T_{i,1}, \ldots, x_{i,p_i}, T_{i,p_i} \vdash_\Sigma e_i : T, x, e' \mapsto e_i', K_i\end{array}\right.}{\begin{array}{l}\Gamma \vdash_\Sigma \mathsf{match}\ x_0\ \mathsf{with}\ c_1(x_{1,1}, \ldots, x_{1,p_1})\langle' \rangle \to e_1 \mid \ldots c_m(x_{m,1}, \ldots, x_{m,p_m})\langle' \rangle \to e_m : T, x, e' \\ \quad \mapsto \mathsf{match}\ x_0\ \mathsf{with}\ c_1(x_{1,1}, \ldots, x_{1,p_1})\langle' \rangle \to e_1' \mid \ldots c_m(x_{m,1}, \ldots, x_{m,p_m})\langle' \rangle \to e_m', \biguplus_i K_i\end{array}} \quad \text{(C-MATCH)}$$

Figure 3.1: CPS transformation for expressions

$$\frac{\Sigma(f) = T_1,\ldots,T_p \to T \qquad x_1 : T_1,\ldots,x_p : T_p \vdash_\Sigma e_f : T, x, \mathsf{unwind\_}T(x,s) \mapsto e'_f, K}{\vdash_\Sigma f(x_1,\ldots,x_p) = e_f \mapsto (\{f'(x_1,\ldots,x_p,s) = e'_f\}, \emptyset, \emptyset) \uplus K} \text{(C-FUNBODY)}$$

$$\frac{\vdash_\Sigma D \mapsto K_1 \qquad \vdash_\Sigma B \mapsto K_2}{\vdash_\Sigma D \text{ and } B \mapsto K_1 \uplus K_2}$$

$$\frac{\vdash_\Sigma B \mapsto K}{\vdash_\Sigma \mathsf{let}\ B \mapsto K}$$

$$\frac{\vdash_\Sigma B \mapsto K_1 \qquad \vdash_\Sigma P \mapsto K_2}{\vdash_\Sigma \mathsf{let}\ B\ P \mapsto K_1 \uplus K_2}$$

$$\frac{\begin{array}{c}\vdash_\Sigma P \mapsto (F,S,R) \\ \forall T \in \mathrm{dom}(R).\ \mathcal{M}_T = \{\mid C(x_1,\ldots,x_n,s') \to \mathsf{cont}C(x,x_1,\ldots,x_n,s') : C \in R(T)\} \\ F_T = \begin{cases} \mathsf{unwind\_}T(x,s) = \mathsf{match}\ s\ \mathsf{with}\ \mathcal{M}_T \mid \mathsf{end} \to x & \text{if } T = T_{\text{final}} \\ \mathsf{unwind\_}T(x,s) = \mathsf{match}\ s\ \mathsf{with}\ \mathcal{M}_T & \text{otherwise} \end{cases} \\ \{F_1,\ldots,F_m\} = F \cup \{F_T : T \in \mathrm{dom}(R)\}\end{array}}{\vdash \Sigma, P \mapsto \Sigma + S[\mathsf{end} \mapsto \mathsf{stack}], \mathsf{let}\ F_1 \text{ and } F_2 \text{ and } \ldots \text{ and } F_m}$$

We put end into nullc. If the original initial function was $f(x_1,\ldots,x_p)$, our new initial function is $f'(x_1,\ldots,x_p,s)$ where $S(s) = \mathsf{null}$.

Figure 3.2: CPS transformation for whole programs

constructors for closures which are called with a value of that type. When we wish to refer to an entire tuple at once, we denote it K. Similarly, we will use $\emptyset$ as a shorthand for $(\emptyset,\emptyset,\emptyset)$. We define a function $\uplus$ for joining this information as

$$(F_1,S_1,R_1) \uplus (F_2,S_2,R_2) = (F_1 \cup F_2, S_1 + S_2, R'),$$

where $+$ joins two disjoint partial maps and

$$R' = [T \mapsto \{C : C \in R_1(T) \text{ or } C \in R_2(T)\} : T \in \mathrm{dom}(R_1) \cup \mathrm{dom}(R_2)].$$

The most important part of the CPS transformation is the C-FUN rule. Our principle requirement for a program in continuation passing style is that all functions must be tail calls. Thus, to allow the continuation expression e' to be executed after the function has been evaluated, the C-FUN rule packages e' into a new continuation function cont$C$ and constructs a closure $C(y_1,\ldots,y_n,s)$ containing the values of the live bound

variables. We call the datatype for these closures `stack` because it replaces the runtime stack.

To see how the continuation is invoked consider the remainder of the transformation, detailed in Figure 3.2. The C-FUNBODY rule transforms the function f with $\mathsf{unwind\_}T(x,s)$ as the continuation expression. The new $\mathsf{unwind\_}T$ family of functions examines the 'stack' s and calls the relevant continuation function with the result from the evaluation of f and the contents of the closure. Note that the closure will always be deallocated; we know that this is safe because the stack discipline ensures that the closure will never be accessed again.

The whole transformation is given as a judgement

$$\vdash \Sigma, P \mapsto \Sigma', P'$$

providing a new set of function signatures as well as the transformed program. A consequence of the transformation is that P' consists of functions in a single mutually-recursive block. This is required because the $\mathsf{unwind\_}T$ functions are called throughout the program, destroying the block structure.

To demonstrate the transformation, let us examine the effect on a simple function.

Example 3.1. Suppose we have a program containing the following function:

```
let pairf(a) = let b = f(a) in (a,b)
```

where

$$\Sigma = \begin{bmatrix} \texttt{f} & \mapsto & T_1 \to T_2 \\ \texttt{pairf} & \mapsto & T_1 \to T_1 \otimes T_2 \\ & \vdots & \end{bmatrix}.$$

We will trace the transformation of `pairf` starting at the function transformation rule, C-FUNBODY. This requires us to find some $e'_{\texttt{pairf}}$ and K such that

$$\mathsf{a} : T_1 \vdash_\Sigma \texttt{let b} = \cdots : T_1 \otimes T_2, x, \mathsf{unwind\_}T_1 \otimes T_2(x,s) \mapsto e'_{\texttt{pairf}}, K.$$

Hence we wish to use C-LET, and need to fulfill its premises. We start with the pairing expression (a,b) because we already know its continuation expression from C-LET:

$$\mathcal{D}_{\mathsf{pair}} = \frac{}{\begin{array}{l}\mathsf{a} : T_1, \mathsf{b} : T_1 \vdash_\Sigma (\mathsf{a},\mathsf{b}), x, \mathsf{unwind\_}T_1 \otimes T_2(x,s) \\ \qquad \mapsto \mathsf{let}\ x = (\mathsf{a},\mathsf{b})\ \mathsf{in}\ \mathsf{unwind\_}T_1 \otimes T_2(x,s), \emptyset\end{array}}\ \text{C-PAIR}$$

The pair expression produces the result for the entire function, so the transformed version must call the next continuation on the stack, which it does via $\mathsf{unwind\_}T_1 \otimes T_2(x,s)$.

Let us call the transformed expression for the pairing e_p. It is now used as the continuation expression for the first subexpression of the let, `f(a)`:

$$\dfrac{K = \left(\mathsf{contC1}(\mathtt{b},\mathtt{a},s) = e_\mathsf{p}, \left[\begin{array}{c}\mathsf{C1} \mapsto (T_1, \mathsf{stack} \to \mathsf{stack}) \\ \mathsf{contC1} \mapsto (T_2, T_1, \mathsf{stack} \to T_{\text{final}})\end{array}\right], [T_2 \mapsto \{\mathsf{C1}\}]\right) \quad \text{C-FUN}}{\dfrac{\mathtt{a} : T_1 \vdash_\Sigma \mathtt{f(a)} : T_2, \mathtt{b}, e_\mathsf{p} \mapsto \mathsf{let}\ s' = \mathsf{C1}(\mathtt{a},s)\ \mathsf{in}\ \mathtt{f}'(\mathtt{a},s'), K \qquad \mathcal{D}_\mathsf{pair}}{\mathtt{a} : T_1 \vdash_\Sigma \mathsf{let}\ \mathtt{b} = \cdots : T_1 \otimes T_2, x, \mathsf{unwind\_}(x,s) \mapsto \mathsf{let}\ s' = \mathsf{C1}(\mathtt{a},s)\ \mathsf{in}\ \mathtt{f}'(\mathtt{a},s'), K}\ \text{C-LET}}$$

Thus the resulting code for `pairf'` is

```
let pairf'(a,s) = let s'= C1(a,s) in f'(a,s')
```

which puts `a` on to the 'stack' with tag C1 and calls the transformed `f'`. When this produces a result, $\mathsf{unwind\_}T_2$ (defined by the last rule in 3.2 using the information in K) will call the continuation contC1,

```
and contC1(b,a,s) = let x=(a,b) in unwind_T1*T2(x,s)
```

with `b` and `a` to form the pair.

This CPS transformation has been implemented as an extension to Jost's `lfd_infer`.

We can now establish some basic properties of the transformed program:

Lemma 3.2. *Every function call in a CPS transformed program is in tail position, except for the 'initial' function call.*

Proof. By induction on the transformation derivation. Only let expressions can introduce a subexpression which is not in tail position, but the only transformation rules which produce a let are the leaf rules (C-UNIT, C-BOOL, C-VAR, C-PAIR, C-INL, C-INR, C-CONSTRUCT and C-FUN), none of which place a function call in the left hand subexpression.

Thus all of the function calls which appear in transformed function bodies are in tail position. This leaves only the initial function call, which is not in tail position by definition. □

Using this we can show that we only need enough 'real' stack space for the current function:

Corollary 3.3. *Under the 'general tail-call optimisation' version of the operational semantics (Section 2.1.2) the stack memory used in evaluating a transformed program is at most* $\max_f(\text{stack}(f))$.

Proof. Let $m = \max_f(\text{stack}(f))$. By Lemma 3.2 all function calls are tail calls except for the initial function. The initial function (say, f_1) requires

$$\text{stack}'(\text{initial}, f_1, \text{false}) = \text{stack}(f_1) \leq m$$

units of stack space. Each subsequent call requires a change of

$$\text{stack}'(f_i, f_{i+1}, \text{true}) = \text{stack}(f_{i+1}) - \text{stack}(f_i)$$

units. Thus the amount of stack space required during the evaluation of f_i is

$$\text{stack}(f_1) + (\text{stack}(f_2) - \text{stack}(f_1)) + \cdots + (\text{stack}(f_i) - \text{stack}(f_{i-1})) = \text{stack}(f_i) \leq m.$$

□

When combined with the Hofmann-Jost analysis to infer a bound on the heap memory this result provides a bound on the total memory usage of the transformed program.

3.2 Correctness of the transformation

We have shown that the transformed program's total memory usage can be analysed, but we still need to establish that it faithfully reproduces the original program's behaviour.

First we must show that we can remove dead variables from the environment of an evaluation because C-FUN only places potentially live variables in the closure.

Lemma 3.4. *Given* $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ *and any* $V \subseteq \text{dom}(S)$ *such that* $\text{FV}(e) \cap V = \emptyset$ *we have*

$$S \backslash V, \sigma \vdash e \rightsquigarrow v, \sigma'.$$

Proof. A straightforward induction on the evaluation shows that the values of V are not used by any rule. □

We are also able to extend the environment and state:

Lemma 3.5. *Given* $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ *and some* S_1, σ_1 *disjoint from the evaluation, then we have*

$$S + S_1, \sigma + \sigma_1 \vdash e \rightsquigarrow v, \sigma' + \sigma_1.$$

Proof. Induction on the evaluation, adding the new environment S_1 and state σ_1 where necessary. As they are disjoint from the given evaluation they do not interfere in any way, and v and σ' are not changed. □

We can now show that the transformation of an expression is correct.

Theorem 3.6. *Suppose we have a typed program and its CPS transformation. For any part of the transformation on an expression*

$$\Gamma \vdash_\Sigma e : T, x, e' \mapsto e'', K$$

if $S, \sigma \vdash e \rightsquigarrow v_0, \sigma_0$ *and* $S[x \mapsto v_0, s \mapsto l], \sigma_0 + \sigma_s \vdash e' \rightsquigarrow v', \sigma'$ *for some* l, σ_s *disjoint from* σ *and* σ_0 *then*

$$S[s \mapsto l], \sigma + \sigma_s \vdash e'' \rightsquigarrow v', \sigma'.$$

Proof. We proceed by induction on the evaluation of the original expression, e.

E-UNIT, E-BOOL, E-VAR, E-PAIR, E-INL, E-INR, E-CONSTRUCT, E-CONSTRUCTN. We have $e'' = \mathsf{let}\ x = e\ \mathsf{in}\ e'$. Hence:

$$\frac{\dfrac{}{S[s \mapsto l], \sigma + \sigma_s \vdash e \rightsquigarrow v_0, \sigma_0 + \sigma_s}\ \text{by Lemma 3.5} \qquad \dfrac{}{S[x \mapsto v_0, s \mapsto l], \sigma_0 + \sigma_s \vdash e' \rightsquigarrow v', \sigma'}\ \text{by hypothesis}}{S[s \mapsto l], \sigma + \sigma_s \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \rightsquigarrow v', \sigma'}\ \text{E-LET}$$

E-FUN. First, consider the invocation of the function in the original program,

$$\frac{[y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p)], \sigma \vdash e_f \rightsquigarrow v_0, \sigma_0}{S, \sigma \vdash f(x_1, \ldots, x_p) \rightsquigarrow v_0, \sigma_0}\ \text{E-FUN}$$

and the transformed program:

$$\mathcal{D}_c = \frac{\sigma'_s = \sigma_s[l' \mapsto (C, S(z_1), \ldots, S(z_n), l)]}{S[s \mapsto l], \sigma + \sigma_s \vdash C(z_1, \ldots, z_n, s) \rightsquigarrow l', \sigma + \sigma'_s}\ \text{E-CONSTRUCT}$$

$$\frac{\mathcal{D}_c \qquad \dfrac{[y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p), s \mapsto l'], \sigma + \sigma'_s \vdash e_{f'} \rightsquigarrow v', \sigma'}{S[s \mapsto l, s' \mapsto l'], \sigma + \sigma'_s \vdash f'(x_1, \ldots, x_p, s') \rightsquigarrow v', \sigma'}\ \text{E-FUN}}{S[s \mapsto l], \sigma + \sigma_s \vdash \mathsf{let}\ s' = C(z_1, \ldots, z_n, s)\ \mathsf{in}\ f'(x_1, \ldots, x_p, s') \rightsquigarrow v', \sigma'}\ \text{E-LET}$$

where the y_i are the names of f's arguments and the z_i are the live variables for the closure.

The body of f', $e_{f'}$ was created from e_f by a transformation of the form:

$$y_1 : T_1, \ldots, y_p : T_p \vdash e_f, x, \mathsf{unwind\_}T(x, s) \mapsto e_{f'}, F$$

To use the induction hypothesis with this transformation we need to derive an evaluation for our new continuation, $\mathsf{unwind\_}T(x,s)$, from our original continuation, e':

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{by hypothesis and Lemma 3.4}}{(S \restriction \mathrm{FV}(e)\backslash\{x,s\})[x \mapsto v_0, s \mapsto l], \sigma_0 + \sigma_s \vdash e' \rightsquigarrow v', \sigma'}}{[x \mapsto v_0, z_1 \mapsto S(z_1), \ldots, z_n \mapsto S(z_n), s' \mapsto l], \sigma_0 + \sigma_s \vdash \mathsf{cont}C(x, z_1, \ldots, z_n, s') \rightsquigarrow v', \sigma'}\ \text{E-FUN}}{[x \mapsto v_0, s \mapsto l'], \sigma_0 + \sigma'_s \vdash \mathsf{match}\ s\ \mathsf{with}\ \cdots \mid C(z_1, \ldots, z_n, s') \to \ldots \rightsquigarrow v', \sigma'}\ \text{E-MATCH}}{[y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p), x \mapsto v_0, s \mapsto l'], \sigma_0 + \sigma'_s \vdash \mathsf{unwind\_}T(x,s) \rightsquigarrow v', \sigma'}\ \text{E-FUN}$$

Thus the induction hypothesis yields the evaluation of $e_{f'}$ required to complete the transformed expression's evaluation, above.

E-LET. The original expression's evaluation takes the form:

$$\dfrac{S, \sigma \vdash e_1 \rightsquigarrow v_1, \sigma_1 \qquad S[x_1 \mapsto v_1], \sigma_1 \vdash e_2 \rightsquigarrow v_0, \sigma_0}{S, \sigma \vdash \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v_0, \sigma_0}\ \text{E-LET}$$

We can apply the induction hypothesis to e_2 using e' as the continuation expression to yield an evaluation for e'_2. Now we apply the induction hypothesis to e_1, using e'_2 as the continuation expression which gives an evaluation for e'', as required.

E-IFTRUE. Consider the evaluation of the original expression:

$$\dfrac{S(x) = \mathsf{true} \qquad S, \sigma \vdash e_t \rightsquigarrow v, \sigma'}{S, \sigma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \rightsquigarrow v, \sigma'}\ \text{E-IFTRUE}$$

The induction hypothesis on e_t yields the evaluation of the e'_t subexpression in the transformed program. It is then sufficient to apply the E-IFTRUE rule again to get the evaluation of e''.

E-IFFALSE, E-MATCHPAIR, E-MATCHINL, E-MATCHINR, E-MATCH, E-MATCH′, E-MATCHN, E-MATCHN′. Similar to E-IFTRUE; apply the induction hypothesis to the subexpression and then use the original evaluation rule. □

Of course, this extends to the whole program:

Corollary 3.7. *Given a typed program and its CPS transformation, for any evaluation of the initial function*

$$S, \sigma \vdash f(x_1, \ldots, x_p) \rightsquigarrow v, \sigma'$$

we have

$$S[s \mapsto \mathsf{null}], \sigma \vdash f'(x_1, \ldots, x_p, s) \rightsquigarrow v, \sigma'$$

in the transformed program.

Proof. The evaluation of the original program begins with E-FUN, so we have

$$[x_1 \mapsto S(x_1), \ldots, x_p \mapsto S(x_p)], \sigma \vdash e_f \rightsquigarrow v, \sigma'.$$

The transformation must include a judgement for the body of f,

$$x_1 : T_1, \ldots, x_p : T_p \vdash_\Sigma e_f : T_{\text{final}}, x, \mathsf{unwind\_}T_{\text{final}}(x, s) \mapsto e_{f'}, K,$$

and we can evaluate the continuation as follows:

$$\dfrac{\dfrac{\dfrac{}{[x \mapsto v, s \mapsto \mathsf{null}], \sigma' \vdash x \rightsquigarrow v, \sigma'}\ \text{E-VAR}}{[x \mapsto v, s \mapsto \mathsf{null}], \sigma' \vdash \mathsf{match}\ s\ \mathsf{with}\ \cdots \mid \mathsf{end} \to x \rightsquigarrow v, \sigma'}\ \text{E-MATCHN}}{[x_1 \mapsto S(x_1), \ldots, x_p \mapsto S(x_p), s \mapsto \mathsf{null}, x \mapsto v], \sigma' \vdash \mathsf{unwind\_}T_{\text{final}}(x, s) \rightsquigarrow v, \sigma'}\ \text{E-FUN}$$

So by Theorem 3.6 we have

$$[x_1 \mapsto S(x_1), \ldots, x_p \mapsto S(x_p), s \mapsto \mathsf{null}], \sigma \vdash e_{f'} \rightsquigarrow v, \sigma'$$

and hence by E-FUN

$$S[s \mapsto \mathsf{null}], \sigma \vdash f'(x_1, \ldots, x_p, s) \rightsquigarrow v, \sigma'.$$ □

3.3 Bounding the original program

An obvious complaint with the above approach is that we require the compiler to use the CPS transformation as part of the compilation. This is inconvenient if we already possess a perfectly good compiler. Moreover, it can be impossible if the target does not allow for general tail call optimisation, as was the case with Java bytecode in the MRG project.

Fortunately, there is a converse result to the correctness theorem which shows that in addition to computing the correct result, we can bound the total space used by the original program using a bound on the heap space of the transformed program.

We defer consideration of tail call optimisation in the evaluation of the original program for now. The transformed program is necessarily 'executed' with tail call optimisation, but as there is no real machine involved we may choose size and stack$'$ to suit our theorem.

Theorem 3.8. *Suppose we have a typed program and its CPS transformation where* $\text{size}(C) = \text{stack}(f)$ *for every continuation closure C introduced in the transformation of the body of* f*. Given the transformation*

$$\Gamma \vdash_\Sigma e : T, x, e' \mapsto e'', K$$

of any expression e in the program, suppose that we have evaluations for e, e' *and* e'' *as per Theorem 3.6, and moreover that we have m,* m' *such that*

$$m, S[s \mapsto l], \sigma + \sigma_s \vdash e'' \rightsquigarrow v', \sigma', m'$$

where we do not count stack space (that is, $\text{stack}(f') = 0$ *for all functions* f' *in the transformed program). Then there exists* m_0 *such that*

$$m + m_s - \text{stack}(g), S, \sigma \vdash e \rightsquigarrow v_0, \sigma_0, m_0 + m_s - \text{stack}(g)$$
$$\text{and} \quad m_0, S[x \mapsto v_0, s \mapsto l], \sigma_0 + \sigma_s \vdash e' \rightsquigarrow v', \sigma', m'$$

where $m_s = \max_f \text{stack}(f)$ *and g is the function containing e.*

Proof. We consider each of the cases from Theorem 3.6 and show that we can determine suitable values for m_0 and other intermediate amounts of memory for e from the derivation for e''. For clarity we omit the environments, state and values from the derivations below.

E-UNIT, E-PAIR, E-VAR, E-PAIR, E-INL, E-INR, E-CONSTRUCT, E-CONSTRUCTN. The evaluation of the transformed program takes the form

$$\dfrac{\dfrac{}{m \vdash e \rightsquigarrow m''} \qquad \begin{array}{c}\vdots\\ m'' \vdash e' \rightsquigarrow m'\end{array}}{m \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \rightsquigarrow m'}\ \text{E-LET-TAIL}$$

for some m''. Taking $m_0 = m''$ and adding $m_s - \text{stack}(g)$ throughout the evaluation of e we get

$$m + m_s - \text{stack}(g) \vdash e \rightsquigarrow m_0 + m_s - \text{stack}(g)$$

and $m_0 \vdash e' \rightsquigarrow m'$ as required.

E-FUN. The derivation of the transformed function call is of the form

$$\dfrac{\dfrac{}{m \vdash C(z_1, \ldots, z_n, s') \rightsquigarrow m_1}\ \text{E-CONSTRUCT} \qquad \dfrac{\begin{array}{c}\vdots\\ m_1 \vdash e_{f'} \rightsquigarrow m'\end{array}}{m_1 \vdash f'(x_1, \ldots, n_p, s') \rightsquigarrow m'}\ \text{E-FUN-TAIL}}{m \vdash \mathsf{let}\ s' = C(z_1, \ldots, z_n, s)\ \mathsf{in}\ f'(x_1, \ldots, x_p, s') \rightsquigarrow m'}\ \text{E-LET-TAIL}$$

with

$$\begin{aligned} m &= m_1 + \text{size}(C) && \text{by E-CONSTRUCT} \\ &= m_1 + \text{stack}(g) && \text{by assumption.} \end{aligned}$$

Recall that $e_{f'}$ is the result of transforming e_f with $\mathsf{unwind\_}T(x,s)$ as the continuation expression. Applying the induction hypothesis to $e_{f'}$ gives us m_0' such that

$$m_1 + m_s - \text{stack}(f) \vdash e_f \rightsquigarrow m_0' + m_s - \text{stack}(f) \quad \text{and} \quad m_0' \vdash \mathsf{unwind\_}T(x,s) \rightsquigarrow m'.$$

The derivation for $\mathsf{unwind\_}T(x,s)$ has the form

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots \\ m_0 \vdash e' \rightsquigarrow m'\end{array}}{m_0 \vdash \mathsf{cont}C(x, y_1, \ldots, y_n, s') \rightsquigarrow m'}\ \text{E-FUN-TAIL}}{m_0' \vdash \mathsf{match}\ s\ \mathsf{with}\ \ldots \rightsquigarrow m'}\ \text{E-MATCH}}{m_0' \vdash \mathsf{unwind\_}T(x,s) \rightsquigarrow m'}\ \text{E-FUN-TAIL}$$

where $m_0 = m_0' + \text{size}(C) = m_0' + \text{stack}(g)$.

Now consider the evaluation of the original expression by applying E-FUN to the judgement for e_f we obtained from the induction hypothesis,

$$\dfrac{m_1 + m_s - \text{stack}(f) \vdash e_f \rightsquigarrow m_0' + m_s - \text{stack}(f)}{m_f \vdash f(x_1, \ldots, x_p) \rightsquigarrow m_f'}\ \text{E-FUN}$$

where

$$\begin{aligned} m_f &= m_1 + m_s = m + m_s - \text{stack}(g), \\ m_f' &= m_0' + m_s = m_0 + m_s - \text{stack}(g). \end{aligned}$$

So,

$$m + m_s - \text{stack}(g) \vdash f(x_1, \ldots, x_p) \rightsquigarrow m_0 + m_s - \text{stack}(g) \quad \text{and} \quad m_0 \vdash e' \rightsquigarrow m'.$$

E-LET. Following the proof of Theorem 3.6 in reverse, we apply the induction hypothesis to e_1 with e_2' as the continuation expression, yielding m_1 such that

$$m + m_s - \text{stack}(g) \vdash e_1 \rightsquigarrow m_1 + m_s - \text{stack}(g) \quad \text{and} \quad m_1 \vdash e_2' \rightsquigarrow m'.$$

Then we apply the induction hypothesis to e_2 with e' as the continuation expression, giving m_0 similarly. Thus

$$\frac{m + m_s - \text{stack}(g) \vdash e_1 \rightsquigarrow m_1 + m_s - \text{stack}(g) \qquad m_1 + m_s - \text{stack}(g) \vdash e_2 \rightsquigarrow m_0 + m_s - \text{stack}(g)}{m + m_s - \text{stack}(g) \vdash \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2 \rightsquigarrow m_0 + m_s - \text{stack}(g)}\ \text{E-LET}$$

and $m_0 \vdash e' \rightsquigarrow m'$ as required.

E-IFTRUE, E-IFFALSE, E-MATCHPAIR, E-MATCHINL, E-MATCHINR, E-MATCH, E-MATCH′, E-MATCHN, E-MATCHN′. As in Theorem 3.6 we apply the induction hypothesis to the subexpression which will be evaluated, and then use the evaluation rule to obtain the result. □

We can also give a bound on the total memory required to evaluate the whole program, starting with the initial function:

Corollary 3.9. *Given the same assumptions as Theorem 3.8 about the program and the* size *and* stack *measures, if we have*

$$m, S[s \mapsto \mathsf{null}], \sigma \vdash f'(x_1, \ldots, x_p, s) \rightsquigarrow v, \sigma', m'$$

then

$$m + m_s, S, \sigma \vdash f(x_1, \ldots, x_p) \rightsquigarrow v, \sigma', m' + m_s,$$

where $m_s = \max_f \text{stack}(f)$.

Proof. The transformed program's evaluation begins with the initial function call:

$$\frac{m, [y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p), s \mapsto \mathsf{null}], \sigma \vdash e_{f'} \rightsquigarrow v, \sigma', m'}{m, S[s \mapsto \mathsf{null}], \sigma \vdash f'(x_1, \ldots, x_p, s) \rightsquigarrow v, \sigma', m'}\ \text{E-FUN-TAIL}$$

The body $e_{f'}$ is e_f CPS transformed with continuation expression $\mathsf{unwind\_}T(x,s)$. Applying Theorem 3.8 we get m_0 such that

$$m + m_s - \text{stack}(f) \vdash e_f \rightsquigarrow m_0 + m_s - \text{stack}(f)$$

and $m_0 \vdash \mathsf{unwind\_}T(x,s) \rightsquigarrow m'$. The evaluation of $\mathsf{unwind\_}T(x,s)$ takes the form

$$\dfrac{\dfrac{\dfrac{}{m' \vdash x \rightsquigarrow m'}\ \text{E-VAR}}{m' \vdash \mathsf{match}\ s\ \mathsf{with}\ \ldots \mathsf{end} \to x \rightsquigarrow m'}\ \text{E-MATCHN}}{m' \vdash \mathsf{unwind\_}T(x,s) \rightsquigarrow m'}\ \text{E-FUN}$$

using $S(s) = \mathsf{null}$, so $m_0 = m'$. Finally, by E-FUN we have

$$\frac{m + m_s - \text{stack}(f), [y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p)], \sigma \vdash e_f \rightsquigarrow v, \sigma', m' + m_s - \text{stack}(f)}{m + m_s, S, \sigma \vdash f(x_1, \ldots, x_p) \rightsquigarrow v, \sigma', m' + m_s}\ \text{E-FUN.}$$

□

Thus we can use Hofmann-Jost on the transformed program to obtain a bound on the *total* memory usage of the original program.

3.4 Tail call optimisation

We can also model the effect of tail call optimisation by adding a special rule for it to the transformation. The idea is to skip the construction of the unnecessary continuation:

$$\frac{}{\Gamma \vdash_\Sigma f(x_1,\ldots,x_p) : T, x, \mathsf{unwind\_}T(x,s) \mapsto f(x_1,\ldots,x_p,s), \emptyset} \text{(C-FUNTAIL)}$$

Before we prove that this rule behaves correctly, we first note the following useful lemma (which is true regardless of whether we include C-FUNTAIL):

Lemma 3.10. *In a transformation derivation the continuation expression is of the form* $\mathsf{unwind\_}T(x,s)$ *iff the original expression is in tail position.*

Proof. By induction on the depth of the CPS transformation. The only rule which can introduce $\mathsf{unwind\_}T(x,s)$ is C-FUNBODY which transforms function definitions. This corresponds exactly to the introduction of a true tail position flag in the operational semantics.

Now suppose the lemma is true for an original expression in the transformation. If it is a leaf expression — the C-UNIT, C-BOOL, C-VAR, C-PAIR, C-INL, C-INR, C-CONSTRUCT, and C-FUN cases — then there are no subexpressions.

For the C-IF, C-MATCHPAIR, C-MATCHSUM and C-MATCH cases all the subexpressions are in tail position iff the current expression is. They are also transformed with the same continuation expression, so the Lemma holds.

The remaining case is C-LET. The e_2 subexpression follows by the same reasoning as the previous case. The e_1 subexpression is not in tail position, and its continuation expression is the result of transforming e_2. However, the result of a transformation is never the continuation expression alone, and we have already remarked that $\mathsf{unwind\_}T(x,s)$ is only introduced as a continuation expression when transforming a function body. Thus the continuation expression for transforming e_1 is not $\mathsf{unwind\_}T(x,s)$. □

Now we can show that the addition of the rule preserves correctness and predicts the memory usage of programs under tail call optimisation.

Theorem 3.11. *With the addition of full tail call optimisation for the original program's evaluation and the* C-FUNTAIL *rule in the transformation,*

1. *the simulation result of Theorem 3.6 and Corollary 3.7, and*
2. *the memory usage result of Theorem 3.8 and Corollary 3.9*

still hold.

Proof. We only need to consider the function application rules in the inductions. By Lemma 3.10 only tail calls are affected by C-FUNTAIL and all other function calls are unchanged. Thus we only consider expressions in tail position where C-FUNTAIL is used. For part 1 note that the original program's evaluation

$$\frac{[y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p)], \sigma \vdash e_f \rightsquigarrow v_0, \sigma_0}{S, \sigma \vdash f(x_1, \ldots, x_p) \rightsquigarrow v_0, \sigma_0}\ \text{E-FUN-TAIL}$$

and the transformed program's (when using E-FUNTAIL)

$$\frac{[y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p), s \mapsto l], \sigma \vdash e_{f'} \rightsquigarrow v', \sigma'}{S[s \mapsto l], \sigma \vdash f'(x_1, \ldots, x_p, s) \rightsquigarrow v', \sigma'}\ \text{E-FUN-TAIL}$$

proceed in the same way because we do not form a new continuation. The result then follows by the induction hypothesis on e_f because it was also transformed with the continuation expression $\mathsf{unwind\_}T(x,s)$.

For part 2, the evaluation of the transformed expression has the form

$$\frac{m \vdash e_{f'} \rightsquigarrow m'}{m \vdash f'(x_1, \ldots, x_p, s) \rightsquigarrow m'}\ \text{E-FUN-TAIL}$$

because $\text{stack}(f') = 0$ by assumption. By the induction hypothesis on e_f there exists m_0 such that

$$m + m_s - \text{stack}(f) \vdash e_f \rightsquigarrow m_0 + m_s - \text{stack}(f)$$
$$\text{and} \quad m_0 \vdash \mathsf{unwind\_}T(x,s) \rightsquigarrow m'.$$

Now, by E-FUN-TAIL the space required for the function call in the original program is m plus

$$(m_s - \text{stack}(f)) + \text{stack}'(g, f, \mathsf{true}) = m_s - \text{stack}(f) + \text{stack}(f) - \text{stack}(g) = m_s - \text{stack}(g)$$

so we have

$$m + m_s - \text{stack}(g) \vdash f(x_1, \ldots, x_p) \rightsquigarrow m_0 + m_s - \text{stack}(g)$$
$$\text{and} \quad m_0 \vdash \mathsf{unwind\_}T(x,s) \rightsquigarrow m',$$

as required. □

3.5 Examples and drawbacks

We now return to the examples from Section 2.2.4.

Example 3.12. Recall the `notlist` function:

```
let notlist l =
  match l with nil' -> nil
             | cons(h,t)' ->
                 let hh = if h then false else true in
                 let tt = notlist t in
                   cons(hh,tt)
```

The CPS transformation places the new list elements on the 'stack' and creates the list as it is unwound:

```
let notlist'(l, s) =
  match l with nil' -> let x = nil in unwind_boollist(x,s)
             | cons(h,t)' ->
                       if h then let hh = false in
                                 let s' = C1(hh,s) in notlist'(t,s')
                            else let hh = true in
                                 let s' = C1(hh,s) in notlist'(t,s')
and contC1(tt,hh,s) = let x = cons(hh,tt) in unwind_boollist(x,s)
and unwind_boollist(x,s) = match s with C1(hh,s') -> contC1(x,hh,s')
                                      | end -> x
```

with

$$\Sigma = \left[\begin{array}{r} \texttt{nil} \mapsto \mathsf{boollist}, \\ \texttt{cons} \mapsto \mathsf{bool}, \mathsf{boollist} \to \mathsf{boollist}, \\ \texttt{end} \mapsto \mathsf{stack}, \\ \texttt{C1} \mapsto \mathsf{bool}, \mathsf{stack} \to \mathsf{stack}, \\ \texttt{notlist}' \mapsto \mathsf{boollist}, \mathsf{stack} \to \mathsf{boollist}, \\ \texttt{contC1} \mapsto \mathsf{boollist}, \mathsf{bool}, \mathsf{stack} \to \mathsf{boollist}, \\ \texttt{unwind\_boollist} \mapsto \mathsf{boollist}, \mathsf{stack} \to \mathsf{boollist} \end{array}\right].$$

The *total* memory usage of `notlist'(l,end)` will be at most

$$|\texttt{l}| \times \max\{\text{size}(\texttt{C1}), \text{size}(\texttt{cons})\} + \max_f(\text{stack}(f)),$$

because the list's contents is built on the 'stack' using `C1`, then the list is constructed as the stack is unwound, while at most $\max_f(\mathsf{stack}(f))$ will be required to store local variables.

If we assume that the sizes of the two data structures are both one, then the Hofmann-Jost analysis yields the signatures

$$\begin{aligned}
\texttt{notlist}' &: \quad \mathsf{boollist}(1), \mathsf{stack}(1), 0 \to \mathsf{boollist}(0), 0 \\
\texttt{contC1} &: \mathsf{boollist}(0), \mathsf{bool}, \mathsf{stack}(1), 1 \to \mathsf{boollist}(0), 0 \\
\texttt{unwind_boollist} &: \quad \mathsf{boollist}(0), \mathsf{stack}(1), 0 \to \mathsf{boollist}(0), 0
\end{aligned}$$

confirming that $|\texttt{l}|$ units of free heap space is required by the transformed program. If we add the largest frame size as per Corollary 3.3 then it agrees with the total usage above. This also bounds the total memory usage of the original program.

We can consider the stack space alone by setting $\mathsf{size}(\texttt{cons})$ to zero. If we assume that the closure's size (that is, `C1`'s size) is equal to $\mathsf{stack}(\texttt{notlist})$ then the stack usage should be

$$|\texttt{l} + 1| \times \mathsf{stack}(\texttt{notlist}).$$

Indeed the signatures are now

$$\begin{aligned}
\texttt{notlist}' &: \quad \mathsf{boollist}(1), \mathsf{stack}(0), 0 \to \mathsf{boollist}(0), 0 \\
\texttt{contC1} &: \mathsf{boollist}(0), \mathsf{bool}, \mathsf{stack}(0), 0 \to \mathsf{boollist}(0), 0 \\
\texttt{unwind_boollist} &: \quad \mathsf{boollist}(0), \mathsf{stack}(0), 0 \to \mathsf{boollist}(0), 0
\end{aligned}$$

showing that $|l|$ heap-allocated frames are required by the transformed program. Following Corollary 3.3 again, we add another frame to get the expected total. Note that the annotation on the stack types is now zero, because we no longer require memory for the allocation of the new list. However, the overall bound on the stack space alone is equal to the bound on total space, because the size of the free 'stack' space is large enough to account for the allocation of the new list on the heap.

Example 3.13. The Hofmann-Jost analysis can be successfully applied to the result of transforming the functional heap sort of Appendix A. Using a reasonable size model for the data structures (one word for each integer or location, plus one for constructor tags where necessary) we obtain a signature for the sorting function of

$$\mathsf{intlist}(2), \mathsf{stack}(\dots), 11 \to \mathsf{intlist}(2), 6.$$

This tells us that $11 + 2 \times |l|$ words of memory is sufficient to sort a list l. (We omit the annotations on the stack type because they do not contribute to the bound on the

free memory required at the start of the evaluation.) However, the bound on the free memory afterwards is 5 words lower. We know from the heap-only analysis that all of the extra heap memory is returned and the stack space must be returned at the end of the evaluation. Thus the analysis is 'losing track' of the 5 words at some point.

Attempting to solve this mystery by examining the types is difficult because all the functions have the same result type, '$\mathsf{boollist}(2), 6$', rather than information about the free memory after *that function* is evaluated. To get this information would require examining the types of the $\mathsf{unwind\_}T$ functions and reasoning about the contents of the closures. Fortunately the solution can be found by examining a similar problem in Example 3.14 below.

Analysing the stack space alone gives a *higher* bound of $11 + 4 \times |l|$. The stack contains many partially broken up data structures and the *total* bound assumes that the space from the deallocated structures can be counted against the stack space used. Thus if stack space and heap space are not immediately interchangeable (as is often the case in practice) more stack space is required. In such situations analysing the heap and stack requirements separately gives a more useful account of memory requirements.

Example 3.14. Recall the `id` function introduced in the discussion on resource polymorphism on page 17, which uses `notlist`:

```
let id l = let notl = notlist l in notlist notl
```

Using the CPS transformation on the entire program the `id` portion becomes

```
let id' l s = let s' = C2(s) in notlist' l s'
and contC2(notl,s) = notlist' notl s
and ... [notlist' as before] ...
and unwind_boollist(x,s) =
  match s with C1(hh,s') -> contC1(x,hh,s')
             | C2(s') -> contC2(x,s')
             | end -> x
```

However, the Hofmann-Jost analysis fails (by producing an infeasible linear program). The first obstacle is that our analysis of the original program in Chapter 2 relied on resource polymorphism. The CPS transformation places all of the functions into a single mutually recursive block, so our resource polymorphism mechanism is no longer applicable. We can overcome this by recalling that resource polymorphism is equivalent to duplicating the function in question, so we use two copies of `notlist`.

Unfortunately, this is still not sufficient for the analysis to produce a bound. Consider the type of the lists produced by each call to `notlist` in the original program. The first is

$$\mathsf{boollist}(\mathrm{size}(\mathtt{cons}))$$

because we require enough potential to allocate the second list. The second type is

$$\mathsf{boollist}(0)$$

because no further allocations are performed, so no potential is required. However, every function which produces a boollist value calls the same unwind_boollist function, forcing the types' annotations in the transformed program to be the same throughout. Thus the generated linear program is unfeasible because $\mathrm{size}(\mathtt{cons}) \neq 0$. That is, the change in potential and the type equality cannot be satisfied simultaneously.

We can work around this problem by duplicating the unwind_boollist function once for each `notlist` function and removing the unused cases from each copy. That is, after duplicating `notlist` we have

```
...
and unwind_boollist(x,s) =
  match s with C1(hh,s') -> contC1(x,hh,s')
             | C2(s')    -> contC2(x,s')
             | C3(hh,s') -> contC3(x,hh,s')
             | end       -> x
```

and we then produce:

```
...
and unwind_boollist1(x,s) =
  match s with C1(hh,s') -> contC1(x,hh,s')
             | C2(s')    -> contC2(x,s')
and unwind_boollist2(x,s) =
  match s with C3(hh,s') -> contC3(x,hh,s')
             | end       -> x
```

These can be given different signatures, one where `x` is given the type $\mathsf{boollist}(\mathrm{size}(\mathsf{cons}))$ and one where it is $\mathsf{boollist}(0)$, as required.

However, further complicating the transformation to recover the power of the original analysis does not seem worthwhile if a more direct approach is successful.

On a more positive note, analysing the stack space alone using the transformed program does yield a bound because no resource polymorphism is required.

The trick of manually partitioning the unwind_T functions also works on the heap sort example, recovering the missing 5 words from the bound on the free memory after evaluation.

3.6 Summary

The CPS transformation does provide useful bounds on total memory usage, both when used as part of the compilation process and when performed purely for the analysis. However, to allow a similar range of programs to be analysed as the heap-only analysis would require a yet more complex transformation to perform the function duplication and partitioning of the stack unwinding functions sufficient to replace resource polymorphism. This additional complexity is undesirable, and coupled with the difficulty in relating the inferred types to the original program we conclude that a more direct analysis is preferable. We pursue this in the following chapters.

Chapter 4

A direct adaption of Hofmann-Jost

Our attempt to avoid changing the analysis itself by considering programs in an intermediate CPS language succeeded in providing stack memory bounds and total memory bounds for some programs, but would require a yet more complex transformation to reach the applicability of the heap-only analysis. Thus we now consider how to adapt the analysis directly, without transforming the program.

The principle used in this chapter is to reflect the changes in stack memory allocation in the operational semantics as changes in the potential in the type system, in the same way that heap allocation is handled in the plain Hofmann-Jost analysis.

4.1 Simple adaption

We briefly consider the operational semantics without tail call optimisation to illustrate the principle without too much notational clutter. First, recall the evaluation and typing rules for constructing a (non-null) data structure:

$$\frac{s = (c, S(x_1), \ldots, S(x_p)) \qquad c \notin \mathsf{nullc} \qquad l \notin \mathrm{dom}(\sigma)}{m + \mathrm{size}(c), S, \sigma \vdash c(x_1, \ldots, x_p) \rightsquigarrow l, \sigma[l \mapsto s], m} \text{ (E-CONSTRUCT)}$$

$$\frac{\begin{array}{c}\Sigma(c_i)[\bar{k}] = T_1, \ldots, T_p, k_i \to ty(\bar{k}) \\ \Phi = \{n \geq \mathrm{size}(c_i) + k_i + n'\}\end{array}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash_{\Sigma,F} c_i(x_1, \ldots, x_p) : ty(\bar{k}), n' \mid \Phi} \text{ (CONSTRUCT)}$$

When typing the same term, $c = c_i$, and so $\mathrm{size}(c_i) = \mathrm{size}(c)$, the size of the allocation. Therefore the typing rule ensures that the overall potential drops by at least the size of the allocation. (The k_i term in the constraint does not change the overall potential, but instead compensates for the enlargement of the data structure.)

We wish to adjust the function application rules FUN and FUNDEF in a similar way to reduce the potential during the function call by the size of the extra stack space required in the evaluation rule:

$$\frac{\begin{array}{c} S(x_1) = v_1 \ \ldots \ S(x_p) = v_p \qquad m, [y_1 \mapsto v_1, \ldots, y_p \mapsto v_p], \sigma \vdash e_f \rightsquigarrow v, \sigma', m' \\ \text{the } y_i \text{ are the symbolic arguments in the definition of } f \end{array}}{m + \text{stack}(f), S, \sigma \vdash f(x_1, \ldots, x_p) \rightsquigarrow v, \sigma', m' + \text{stack}(f)} \text{ (E-FUN)}$$

The existing FUNDEF rule from Section 2.2.2 is:

$$\frac{\begin{array}{c} f \notin F \qquad \Sigma(f) = T_1, \ldots, T_p, k \to T, k' | \Phi' \\ \Phi = \{n \geq k, n - k + k' \geq n'\} \end{array}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash_{\Sigma, F} f(x_1, \ldots, x_p) : T, n' \mid \Phi} \text{ (FUNDEF)}$$

The function signature $\Sigma(f)$ tells us the potential required to execute the function body,

$$\sum_{i=1}^{p} \Upsilon(\sigma, S(x_i), T_i) + k,$$

and the potential subsequently produced, $\Upsilon(\sigma', v, T) + k'$. The fixed amounts k and k' in the signature are related to those in the judgement, n and n', by the constraints Φ. We will change these constraints to require $\text{stack}(f)$ more units of potential, and 'release' it again afterwards. Thus the first constraint becomes $n \geq k + \text{stack}(f)$. The second is unchanged because

$$n - (k + \text{stack}(f)) + (k' + \text{stack}(f)) = n - k + k',$$

that is, the stack required and released cancel out. We treat FUN in the same way, giving us the new rules:

$$\frac{\begin{array}{c} f \in F \qquad \Sigma(f) = T'_1, \ldots, T'_p, k \to T', k' | \Phi' \qquad \rho(T'_i) = T_i \qquad \rho(T') = T \\ \Phi = \rho(\Phi') \cup \{n \geq \rho(k) + \text{stack}(f), n - \rho(k) + \rho(k') \geq n'\} \end{array}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash_{\Sigma, F} f(x_1, \ldots, x_p) : T, n' \mid \Phi} \text{ (FUN}')$$

$$\frac{\begin{array}{c} f \notin F \qquad \Sigma(f) = T_1, \ldots, T_p, k \to T, k' | \Phi' \\ \Phi = \{n \geq k + \text{stack}(f), n - k + k' \geq n'\} \end{array}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash_{\Sigma, F} f(x_1, \ldots, x_p) : T, n' \mid \Phi} \text{ (FUNDEF}')$$

Example 4.1. Consider the analysis of the `notlist` function from Section 2.2.4 (page 25). The recursive function call is now typed as

$$\frac{\Phi_1 = \{n_4 \geq n_1 + \text{stack}(\texttt{notlist}), n_4 - n_1 + n_2 \geq n_5\}}{\texttt{t} : \text{boollist}(k_1), n_4 \vdash_{\Sigma, \emptyset} \texttt{notlist t} : \text{boollist}(k_2), n_5 \mid \Phi_1} \text{ FUNDEF}'$$

with the remainder of the derivation the same as before. The minimal solution will have the form

$$n_1 = n_2 = 0,\ n_3 = n_4 = n_5 = k_1 = \max\{\mathrm{stack}(\texttt{notlist}), \mathrm{size}(\texttt{cons})\},\ k_2 = 0,$$
$$\texttt{notlist} : \mathsf{boollist}(\max\{\mathrm{stack}(\texttt{notlist}), \mathrm{size}(\texttt{cons})\}), 0 \to \mathsf{boollist}(0), 0.$$

The maximum is taken because the stack space is returned before the allocation occurs. As usual, if we want independent heap and stack bounds then we can perform the analysis twice, setting stack(`notlist`) and size(`cons`) equal to zero in turn.

Note that the function signature does not include the stack frame from the initial call to `notlist`. This is included in the typing of the caller. For a bound on the whole program (that is, a call to the 'initial function' f) we add stack(f) to the bound from the function signature, in accordance with the FUN typing rule.

Example 4.2. The function

```
let every l = match l with nil' -> true
                         | cons(h,t)' -> if h then every t
                                         else false
```

computes the conjunction of all the elements of a boolean list. Without tail call optimisation each recursive `every t` call requires stack(`every`) units of free space:

$$\mathcal{D}_{\mathsf{cons}} = \dfrac{\dfrac{}{\texttt{t} : \mathsf{boollist}(k), n_1 \vdash_{\Sigma,\emptyset} \texttt{every t} : \mathsf{bool}, n' \mid \Phi_1}\ \textsc{FunDef}' \quad \dfrac{}{\texttt{t} : \mathsf{boollist}(k), n_1 \vdash_{\Sigma,\emptyset} \texttt{false} : \mathsf{bool}, n' \mid \Phi_2}\ \textsc{Bool}}{\texttt{h} : \mathsf{bool}, \texttt{t} : \mathsf{boollist}(k), n_1 \vdash_{\Sigma,\emptyset} \texttt{if} \cdots : \mathsf{bool}, n' \mid \Phi_3}\ \textsc{If}$$

$$\dfrac{\dfrac{\dfrac{}{\cdot, n \vdash_{\Sigma,\emptyset} \texttt{true} : \mathsf{bool}, n' \mid \Phi_4}\ \textsc{Bool}}{\cdot \mid \mathsf{boollist}(k), n \vdash_{\Sigma,\emptyset} \texttt{nil}' \to \cdots : \mathsf{bool}, n' \mid \Phi_4}\ \textsc{Case}' \quad \dfrac{\mathcal{D}_{\mathsf{cons}}}{\cdot \mid \mathsf{boollist}(k), n \vdash_{\Sigma,\emptyset} \texttt{cons(h,t)}' \to \cdots : \mathsf{bool}, n' \mid \Phi_5}\ \textsc{Case}'}{\texttt{l} : \mathsf{boollist}(k), n \vdash_{\Sigma,\emptyset} \texttt{match l} \cdots : \mathsf{bool}, n' \mid \Phi_6}\ \textsc{Match}$$

$$\Phi_1 = \{n_1 \geq n + \mathrm{stack}(\texttt{every})\},\ \Phi_2 = \{n_1 \geq n'\},\ \Phi_3 = \Phi_1 \cup \Phi_2,$$
$$\Phi_4 = \{n \geq n'\},\ \Phi_5 = \Phi_3 \cup \{n_1 = n + k\},\ \Phi_6 = \Phi_4 \cup \Phi_5,$$

with a minimal solution of

$$n = n' = 0,\ n_1 = k = \mathrm{stack}(\texttt{every}),$$
$$\texttt{every} : \mathsf{boollist}(\mathrm{stack}(\texttt{every})), 0 \to \mathsf{bool}, 0.$$

Due to the lack of tail call optimisation, this linear memory bound from the analysis is exactly the amount required according to the operational semantics.

4.2 Tail call optimisation

We can apply the same principle with the tail call optimisation semantics from Section 2.1.2. Recall the function application rule:

$$\frac{S(x_1) = v_1 \ \ldots \ S(x_p) = v_p \qquad m, [y_1 \mapsto v_1, \ldots, y_p \mapsto v_p], \sigma \vdash^{f,\mathsf{true}} e_f \rightsquigarrow v, \sigma', m' \qquad \text{the } y_i \text{ are the symbolic arguments in the definition of } f}{m + \mathrm{stack}'(g,f,t), S, \sigma \vdash^{g,t} f(x_1, \ldots, x_p) \rightsquigarrow v, \sigma', m' + \mathrm{stack}'(g,f,t)} \text{(E-FUN-TAIL)}$$

The amount of allocated memory now changes by $\mathrm{stack}'(g,f,t)$, which also depends on the tail position part of the judgement, t. Thus in addition to changing the function application rules in the type system we need to add the tail position marking that is present in the operational semantics to the typing rules. In particular, the LET and function definition rules need to set the tail position appropriately. These rules now become:

$$\frac{f \in F \qquad \Sigma(f) = T_1', \ldots, T_p', k \to T', k' | \Phi' \qquad \rho(T_i') = T_i \qquad \rho(T') = T \qquad \Phi = \rho(\Phi') \cup \{n \geq \rho(k) + \mathrm{stack}'(g,f,t), n - \rho(k) + \rho(k') \geq n'\}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash^{g,t}_{\Sigma,F} f(x_1, \ldots, x_p) : T, n' \mid \Phi} \text{(FUN-TAIL)}$$

$$\frac{f \notin F \qquad \Sigma(f) = T_1, \ldots, T_p, k \to T, k' | \Phi' \qquad \Phi = \{n \geq k + \mathrm{stack}'(g,f,t), n - k + k' \geq n'\}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash^{g,t}_{\Sigma,F} f(x_1, \ldots, x_p) : T, n' \mid \Phi} \text{(FUNDEF-TAIL)}$$

$$\frac{\Gamma_1, n \vdash^{f,\mathsf{false}}_{\Sigma,F} e_1 : T_0, n_0 \mid \Phi_1 \qquad \Gamma_2, x : T_0, n_0 \vdash^{f,t}_{\Sigma,F} e_2 : T, n' \mid \Phi_2}{\Gamma_1, \Gamma_2, n \vdash^{f,t}_{\Sigma,F} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T, n' \mid \Phi_1 \cup \Phi_2} \text{(LET-TAIL)}$$

$$\frac{\Sigma(f) = T_1, \ldots, T_p, k \to T, k' | \Phi \qquad x_1 : T_1, \ldots, x_p : T_p, k \vdash^{f,\mathsf{true}}_{\Sigma,F} e_f : T, k' \mid \Phi}{\vdash_{\Sigma,F} f(x_1, \ldots, x_p) = e_f \Rightarrow \{f\}}$$

We defer the formal soundness proof until the next chapter because it is a consequence of the theorem for the extended type system presented there. Informally, we have extended both the operational semantics and the Hofmann-Jost system with extra costs in the same way. This close correspondence preserves the soundness of the type system and thus the inferred bounds.

It would also be sound to use a more conservative (that is, larger) stack' in the analysis than the operational semantics. This can allow for variations between implementations, or minor optimisations that are not modelled by the analysis. Moreover, if

we fix $\text{stack}'(f,g,t) = \text{stack}(g)$ we get the analysis without tail call optimisation from the previous section.

Example 4.3. The `notlist` function's typing derivation now has the form

$$\mathcal{D}_1 = \dfrac{\dfrac{}{- \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \texttt{false} : -}\ \textsc{Bool} \qquad \dfrac{}{- \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \texttt{true} : -}\ \textsc{Bool}}{- \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \texttt{if} \cdots : -}\ \textsc{If}$$

$$\mathcal{D}_2 = \dfrac{\dfrac{}{- \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \texttt{notlist t} : -}\ \textsc{FunDef-Tail} \qquad \dfrac{}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{cons} \cdots : -}\ \textsc{Construct}}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{let tt} \cdots : -}\ \textsc{Let-Tail}$$

$$\dfrac{\dfrac{\dfrac{}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{nil} : -}\ \textsc{Construct}}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{nil}' \to \texttt{nil} : -}\ \textsc{Case}' \qquad \dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{let hh} \cdots : -}\ \textsc{Let-Tail}}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{cons(h,t)}' \to \cdots : -}\ \textsc{Case}'}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{match} \cdots : -}\ \textsc{Match}$$

$$\dfrac{\dfrac{}{\vdash_{\Sigma,\emptyset} \texttt{notlist l} = \ldots \Rightarrow \{\texttt{notlist}\}}}{\vdash_{\Sigma,\emptyset} \texttt{let notlist l} = \ldots}$$

The recursive call is not in tail position, so

$$\Phi_1 = \{n_4 \geq n_1 + \text{stack}'(\texttt{notlist}, \texttt{notlist}, \mathsf{false}), n_4 - n_1 + n_2 \geq n_5\},$$

and $\text{stack}'(\texttt{notlist}, \texttt{notlist}, \mathsf{false}) = \text{stack}(\texttt{notlist})$. Therefore the constraints are the same as for the simple adaption, and we get the same total and stack space bounds.

Example 4.4. The `every` function's typing derivation now has the form

$$\dfrac{\dfrac{\dfrac{}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{true} : - \mid -}\ \textsc{Bool}}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{nil}' \to \cdots : - \mid -}\ \textsc{Case}' \qquad \dfrac{\dfrac{}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{every t} : - \mid -}\ \textsc{FunDef-Tail} \qquad \dfrac{}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{false} : - \mid -}\ \textsc{Bool}}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{if} \cdots : - \mid -}\ \textsc{If}}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{cons(h,t)}' \to \cdots : - \mid -}\ \textsc{Case}'$$

$$\dfrac{\qquad}{- \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{match l} \cdots : - \mid -}\ \textsc{Match}$$

Every expression, including the recursive call, is in tail position. So under the reasonable assumption that $\text{stack}'(\texttt{every}, \texttt{every}, \mathsf{true}) = 0$, solving the resulting constraints will give the type signature

$$\texttt{every} : \mathsf{boollist}(0), 0 \to \mathsf{bool}, 0.$$

Thus any call to `every` requires only enough memory for the stack frame for the initial call to `every`; namely, $\text{stack}(\texttt{every})$ units of space.

Example 4.5. The total memory usage of the `id` function from Example 3.14 can now be analysed directly (in contrast to our attempt with the CPS transform). The resource polymorphism allows the type signatures for the two `notlist` calls to differ. In an optimal solution to the generated linear program these will be

$$\begin{aligned}
&\texttt{notlist} : \mathsf{boollist}(k_1), 0 \to \mathsf{boollist}(k_2), 0 \\
&\texttt{notlist} : \mathsf{boollist}(k_2), 0 \to \mathsf{boollist}(0), 0 \\
&\quad \text{where } k_2 = \max\{\mathrm{stack}(\texttt{notlist}), \mathrm{size}(\texttt{cons})\} \\
&\qquad\quad k_1 = \max\{\mathrm{stack}(\texttt{notlist}), \mathrm{size}(\texttt{cons}) + k_2\} = \mathrm{size}(\texttt{cons}) + k_2,
\end{aligned}$$

because at the first call we must have enough potential to perform the recursive call ($\mathrm{stack}(\texttt{notlist})$), then allocate the new list element ($\mathrm{size}(\texttt{cons})$) *and* provide enough potential through the new list's type annotation for the second call (k_2).

Thus `id` will be given the signature

$$\texttt{id} : \mathsf{boollist}(k_1), \mathrm{stack}(\texttt{notlist}) \to \mathsf{boollist}(0), \mathrm{stack}(\texttt{notlist}),$$

and so evaluating `id l` will require $k_1 \times |\texttt{l}| + \mathrm{stack}(\texttt{notlist}) + \mathrm{stack}(\texttt{id})$ units of space, which is a tight bound on the total space required.

4.3 Limitations

The above examples show the analysis performing well in a few simple functions. To gain a better understanding of how well the analysis performs in general we need to consider features of programs that are not well represented by those examples. Three classes which we look at here are programs for which linear bounds on stack space are necessarily overestimates; functions which build their results during recursion (using an accumulator); and larger programs composed from functions for which we can find reasonable bounds on their own.

Note that the limitations we encounter also affect the analysis using CPS transformation from the previous chapter. The CPS method of obtaining stack bounds uses the Hofmann-Jost analysis unchanged, which shows that these limitations also affect the analysis of heap space with Hofmann-Jost where stack-like allocation of data structures occurs.

Sub-linear bounds are often the case for the manipulation of tree structures (for example, the heap in the heap sort example, or the handling of an abstract syntax tree

for a language), and require stack space in proportion to the depth of the data structure. Such bounds cannot be expressed in our current analysis because the potential functions use total sizes and the typing rules work on per-element amounts of potential, implicitly requiring us to measure total sizes. We will consider how to lift these restrictions in Chapter 6.

To examine functions which use accumulators we consider the classic reverse-and-append function:

Example 4.6. The function

```
let revapp(r,a) = match r with nil -> a
                              | cons(h,t) -> let a' = cons(h,a)
                                            in revapp(t,a')
```

reverses `r` and appends the result to `a`. The original `r` is deallocated in the process. The function can be typed by the following derivation (we omit the g, Σ, F from $\vdash^{g,t}_{\Sigma,F}$ and present the constraints afterwards for clarity):

$$\mathcal{D}_{\texttt{revapp}} = \frac{}{\texttt{t} : T\mathsf{list}(k_r), \texttt{a}' : T\mathsf{list}(k_a), n_2 \vdash^{\mathsf{true}} \texttt{revapp} \cdots : T\mathsf{list}(k_a), n'}\ \textsc{FunDef-Tail}$$

$$\mathcal{D}_{\mathsf{cons}} = \dfrac{\dfrac{\dfrac{}{\texttt{h} : T, \texttt{a} : T\mathsf{list}(k_a), n_1 \vdash^{\mathsf{false}} \texttt{cons} \cdots : T\mathsf{list}(k_a), n_2}\ \textsc{Construct} \qquad \mathcal{D}_{\texttt{revapp}}}{\texttt{h} : T, \texttt{t} : T\mathsf{list}(k_r), \texttt{a} : T\mathsf{list}(k_a), n_1 \vdash^{\mathsf{true}} \texttt{let} \cdots : T\mathsf{list}(k_a), n'}\ \textsc{Let-Tail}}{\texttt{a} : T\mathsf{list}(k_a), n \mid T\mathsf{list}(k_r) \vdash^{\mathsf{true}} \texttt{cons}(\texttt{h}, \texttt{t}) \to \cdots : T\mathsf{list}(k_a), n'}\ \textsc{Case}$$

$$\dfrac{\dfrac{}{\texttt{a} : T\mathsf{list}(k_a), n \vdash^{\mathsf{true}} \texttt{a} : T\mathsf{list}(k_a), n'}\ \textsc{Var}}{\texttt{a} : T\mathsf{list}(k_a), n \mid T\mathsf{list}(k_r) \vdash^{\mathsf{true}} \texttt{nil} \to \texttt{a} : T\mathsf{list}(k_a), n'}\ \textsc{Case} \qquad \mathcal{D}_{\mathsf{cons}}}{\texttt{r} : T\mathsf{list}(k_r), \texttt{a} : T\mathsf{list}(k_a), n \vdash^{\mathsf{true}} \texttt{match} \cdots : T\mathsf{list}(k_a), n'}\ \textsc{Match}$$

where

$$\Sigma(\texttt{notlist}) = T\mathsf{list}(k_r), T\mathsf{list}(k_a), n \to T\mathsf{list}(k_a), n'.$$

The constraints accumulated from the typing rules are:

$$\begin{aligned}
&\{n \geq n', && \text{from VAR}\\
&\ n_1 = n + k_r + \mathrm{size}(\texttt{cons}), && \text{from CASE for cons}\\
&\ n_1 \geq \mathrm{size}(\texttt{cons}) + k_a + n_2, && \text{from CONSTRUCT}\\
&\ n_2 \geq n + \mathrm{stack}'(\texttt{revapp}, \texttt{revapp}, \mathsf{true}), \\
&\ n_2 \geq n'\} && \Big\} \text{ from FUNDEF-TAIL}
\end{aligned}$$

Note that the function call does not require any constraints on k_r or k_a because the annotations in the context match those in the function signature. In general we would add weakening of the annotations using SHARE where this is not the case.

If we allow tail call optimisation the function runs in-place, with

$$n = n' = n_2 = 0,\ n_1 = \text{size}(\texttt{cons}) + k_r,\ k_r = k_a \quad \text{for any } k_a.$$

Now suppose that we have no tail call optimisation, so $\text{stack}'(\texttt{revapp}, \texttt{revapp}, \texttt{true}) = \text{stack}(\texttt{revapp})$. The only way to express the (linear) cost of the recursive function calls is the potential of the list k_r, via n_1 and n_2. During evaluation, the stack memory is — as always — returned afterwards, but note that the construction of the new list cell occurs *before* the function call. Hence the potential for the result, k_a, cannot include the potential for the function call, n_2, because of the constraint from CONSTRUCT. Solutions thus take the form

$$n = n' = 0,\ n_1 = \text{size}(\texttt{cons}) + k_r,\ n_2 = \text{stack}(\texttt{revapp}),\ k_r = k_a + \text{stack}(\texttt{revapp}) \text{ for any } k_a,$$

so we require at least $(\text{stack}(\texttt{revapp}) + k_a) \times |\texttt{r}| + k_a \times |\texttt{a}|$ units of stack space, but our bound on the amount afterwards is only k_a times the length of the result. The analysis 'loses track' of the other $\text{stack}(\texttt{revapp}) \times |\texttt{r}|$ units of space.

Accumulating parameters are typically used to facilitate tail recursion. This means that this problem rarely occurs, and we defer a full treatment of accumulating parameters to further work (see Section 9.1.3). However, the extension presented in the next chapter will aid some examples.

Finally, we consider what happens when the analysis is applied to larger programs. One limitation is that we only ever sum the potential for variables in the typing context. For instance, if we have a context

$$a : \mathsf{boollist}(k_a), b : \mathsf{boollist}(k_b)$$

the resulting bound will be of the form $k_a \times |a| + k_b \times |b|$[1]. However, if we call a function which performs recursion over a, and afterwards one which recurses over b, then the *real* stack space requirements will be of the form $\max\{k_a \times |a|, k_b \times |b|\}$. We will consider expressing such bounds precisely in Chapter 6, along with the treatment of bounds which use the depth of data structures.

[1] Finding an optimal solution for the generated linear program may implicitly assign the maximum of some fixed values to k_a or k_b, as in Example 4.5. But we cannot obtain bounds which include the maximum of the size of two *variables* such as $|a|$ and $|b|$ in the present system.

There is another limitation on the form of the bounds which affects larger programs:

Example 4.7. The heap sort program uses a tail recursive list length function to help build the heap. If we change it to a version which is not tail recursive then the stack bound is overestimated. To see why, consider the function's signature:

$$\texttt{length} : \mathsf{intlist}(k), n \to \mathsf{int}, n'$$

Given a list l, the stack memory bound will have the form $k \times |l| + n$, and choosing $k = \mathrm{stack}(\texttt{length})$ is sufficient to satisfy the constraints. However, the bound on the free memory *afterwards* can only be a fixed amount, n', because no potential is given to integer values. Thus there is no way to express the fact that the $k \times |l|$ units of memory are free again, so the analysis 'loses track' of it. This results in an overestimate for the whole heap sort implementation because the reuse of that stack space cannot be taken into account.

We will consider this problem in more detail in the next chapter.

Chapter 5

Accounting for stack space reuse

At the end of the previous chapter we noted that the analysis could not express some bounds on the free stack memory after evaluating a function. In particular, the 'potential' functions that give the bound are defined only in terms of the result's size. This is a problem when the stack memory used cannot be expressed in terms of the result's size alone, or we wish to assign the potential to a later use of one of the arguments. In this chapter we construct an extended version of the space usage analysis where the potential after evaluation can be given in terms of the size of the result *and* the arguments.

5.1 Motivation

The temporary nature of stack memory usage leads to a loss of precision in the analysis presented in Chapter 4. To understand the problem we consider a pair of examples. First, we revisit the list length function from the last chapter.

(In these examples we estimate only the stack memory because the problem affects stack space analysis more acutely, so let $\text{size}(c) = 0$ and $\text{stack}(f) = 1$ for all c, f for these examples. We will show that the extended analysis is sound for both stack and heap memory bounds, however. The extension we will present can also provide some benefit when heap memory is used in a stack-like fashion.)

Example 5.1. Consider the non-tail-recursive list length function,

```
let length l = match l with nil' -> 0
                          | cons(h,t)' -> let n = length t in 1+n
```

$$\texttt{length} : \mathsf{intlist}(k), n \to \mathsf{int}, n' \mid \{k \geq 1, n \geq n'\}$$

where one stack frame per element (i.e. $|\texttt{l}|$ frames) is required. This stack memory is free again after the function returns, but this is not reflected in the function's annotated signature — there is no annotation on the result's type that is capable of representing $|\texttt{l}|$ frames. Should we attempt to use the function twice on the same argument,

```
let twicelength l = let n1 = length l in
                    let n2 = length l in n1+n2
```

then the same stack memory suffices for both calls to `length`. However, the SHARE rule is used to perform contraction of `l` in the typing, as follows:

$$\frac{\texttt{l}_1 : \mathsf{intlist}(k_1), \texttt{l}_2 : \mathsf{intlist}(k_2), n \vdash^{\mathsf{true}}_{\Sigma,F} \begin{array}{l}\texttt{let n1 = length l}_1\ \texttt{in}\\ \texttt{let n2 = length l}_2\ \texttt{in n1+n2} : \mathsf{int}, n'\end{array} \mid \Phi}{\texttt{l} : \mathsf{intlist}(k), n \vdash^{\mathsf{true}}_{\Sigma,F} \begin{array}{l}\texttt{let n1 = length l in}\\ \texttt{let n2 = length l in n1+n2} : \mathsf{int}, n'\end{array} \mid \Phi \cup \{k = k_1 + k_2\}}\ \textsc{Share}$$

This sums the memory requirements for *each* use of l, so the resulting function signature is

$$\texttt{twicelength} : \mathsf{intlist}(k), n \to \mathsf{int}, n' \mid \{k \geq 2, n \geq 1, n \geq n'\},$$

giving a bound of $2 \times |\texttt{l}| + 1$. So the best stack memory bound on `twicelength` is almost twice the actual usage, and again the annotated function signature cannot indicate that the stack memory used is free again after the function returns.

We also need to consider more subtle cases where new data structures are created and they require enough potential to account for memory requirements of later stages of the program.

Example 5.2. The function

```
let andlists (l1, l2) = match l1 with nil' -> nil | cons(h1,t1)' ->
                        match l2 with nil' -> nil | cons(h2,t2)' ->
    let h = if h1 then h2 else false in
    let t = andlists(t1, t2) in
      cons(h,t)
```

computes the pairwise boolean 'and' of two lists. The size of either list would be a reasonable upper bound on the number of stack frames required because the actual number used is the size of the shorter list. The function signature reflects this estimate by requiring a non-zero annotation on at least one of the two arguments:

$$\texttt{andlists} : \mathsf{boollist}(k_1), \mathsf{boollist}(k_2), n \to \mathsf{boollist}(k_1 + k_2), n' \mid \{k_1 + k_2 \geq 1, n \geq n'\}.$$

Note that we get a reasonable estimate on the free memory afterwards from the $k_1 + k_2$ annotation on the result.

Now consider using `andlists` twice, with the same first argument:

```
let andlists2 (l1,l2,l3) = let r1 = andlists (l1,l2) in
                           let r2 = andlists (l1,l3) in (r1, r2)
```

The actual stack bound is $\min\{|\texttt{l1}|, \max\{|\texttt{l2}|, |\texttt{l3}|\}\} + 1$ frames, so one bound we might expect the analysis to be able to produce is $|\texttt{l1}| + 1$. However, the signature we obtain is:

$$\begin{aligned}\texttt{andlists2}: \quad & \mathsf{boollist}(k_{l1} + k_{l1'}), \mathsf{boollist}(k_{l2}), \mathsf{boollist}(k_{l3}), n \to \\ & \mathsf{boollist}(k_{r1}) \otimes \mathsf{boollist}(k_{r2}), n' \\ & \mid \{k_{l1} + k_{l2} \geq 1,\ k_{l1'} + k_{l3} \geq 1,\ k_{l1} + k_{l2} \geq k_{r1},\ k_{l1'} + k_{l3} \geq k_{r2},\ n \geq 1,\ n \geq n'\}\end{aligned}$$

because `l1`'s annotation $k_{l1} + k_{l1'}$ is split between its two uses by the SHARE rule. So while $|\texttt{l1}| + 1$ is a bound on the actual stack usage, the best bound from the typing is $2 \times |\texttt{l1}| + 1$.

Ideally, we ought to be able to reuse the potential from k_{l1} in place of the extra $k_{l1'}$, but with the present system we can only assign the potential to the result, in the form of the k_{r1} annotation.

There are two approaches to extending the potential functions to express tighter bounds for these situations. We could annotate more types and extend the potential function Υ accordingly. In particular, we could annotate the int types and assign values potential proportional to their magnitude. This would enable us to express a tight bound on the stack memory after evaluating `length l`. However, it would not help with `twicelength` because the analysis does not attempt to capture the size relationship between the integer `n1` and the list `l`. Thus assigning potential to `n1` does not help with the second use of `l`. Moreover, it does not help with the `andlists` example at all. Hence, we leave this idea to future work.

Instead, let us consider extending the calculation of the post-evaluation bounds by including the arguments as well as the result. The simplest form of this would be to restrict our attention to stack space and note that we always get back all the free stack memory required for any function call, and so we can simply reuse the old assignment of potential.

For instance, in this hypothetical analysis we could type the two function applications in `twicelength` in the same way by reusing the original assignment of potential

to `l`:

$$\frac{\begin{array}{c}\texttt{l} : \mathsf{intlist}(k), n \vdash^{\mathsf{false}}_{\Sigma,F} \texttt{length l} : \mathsf{int}, n_0 \,|\, \Phi_1 \\ \texttt{l} : \mathsf{intlist}(k), \texttt{n1} : \mathsf{int}, n \vdash^{\mathsf{true}}_{\Sigma,F} \texttt{let n2} = \cdots : \mathsf{int}, n' \,|\, \Phi_2\end{array}}{\texttt{l} : \mathsf{intlist}(k), n \vdash^{\mathsf{true}}_{\Sigma,F} \texttt{let n1} = \cdots : \mathsf{int}, n' \,|\, \Phi_1 \cup \Phi_2}\ \textsc{Hyp-Let}$$

Note that the potential in the result of the first `length l`, n_0, is ignored. Instead we use the original assignment to `l` and n (in contrast to our previous typing using SHARE). We could then get the signature

$$\texttt{twicelength} : \mathsf{intlist}(1), 1 \to \mathsf{int}, 1,$$

which means a bound of $|\texttt{l}| + 1$ for `twicelength l`, which is the actual number of stack frames required.

When typing `andlists2` in such a system, however, we may need to assign potential to the result to satisfy some later use. The two calls to `andlists` could be typed

$$\texttt{l1} : \mathsf{boollist}(1), \texttt{l2} : \mathsf{boollist}(0), 1 \vdash^{\mathsf{false}}_{\Sigma,F} \texttt{andlists(l1,l2)} : \mathsf{boollist}(0), 1$$
$$\texttt{l1} : \mathsf{boollist}(1), \texttt{l3} : \mathsf{boollist}(0), 1 \vdash^{\mathsf{false}}_{\Sigma,F} \texttt{andlists(l1,l3)} : \mathsf{boollist}(1), 1$$

reusing the old assignment of potential to `l1` after the first call, but passing it on to `r2` on the second. This suggests that during inference we would need to make a binary decision at each function application; namely whether to use the old assignment of potential or to assign it to the result instead. This may require an exponential search for the optimal stack memory bound. Also, it forbids splitting the potential between the arguments and the result.

Thus we opt for a more complex system, which uses extra annotations to assign potential to the arguments after evaluation, while still producing constraints which form a linear program and so avoid the binary choices. We also retain the ability to bound the heap space usage in the system because we do not need to assume that allocation follows a stack discipline.

5.2 The ‘give-back’ analysis

We extend the analysis from Chapter 4 by adding an extra *give-back* annotation for each data type annotation in the original system. These new annotations represent the potential assignment to variables *after* the evaluation of the expression. To illustrate

the idea, consider the examples above. For the list length functions in Example 5.1 we will obtain the function signatures

$$\texttt{length} : \mathsf{intlist}(1 \rightsquigarrow 1), 0 \to \mathsf{int}, 0$$
$$\texttt{twicelength} : \mathsf{intlist}(1 \rightsquigarrow 1), 1 \to \mathsf{int}, 1$$

which mean that each function requires one stack frame per list element (plus an extra frame for `twicelength`), *and* that same amount of potential is available for later uses of the list. In particular, both uses of `l` in `twicelength` use the same potential, because the signature of `length` says that we can reuse it.

For Example 5.2 there are several possibilities for the typing, depending upon the requirements for later parts of the program. For each case we will show the typing judgements for the two `andlists` calls and the corresponding function signature for `andlists2`. One choice is to reassign all of the potential to the original list:

$$\texttt{l1} : \mathsf{boollist}(1 \rightsquigarrow 1), \texttt{l2} : \mathsf{boollist}(0 \rightsquigarrow 0), 1 \vdash^{\mathsf{false}}_{\Sigma,F} \texttt{andlists}(\texttt{l1},\texttt{l2}) : \mathsf{boollist}(0 \rightsquigarrow 0), 1$$
$$\texttt{l1} : \mathsf{boollist}(1 \rightsquigarrow 1), \texttt{l3} : \mathsf{boollist}(0 \rightsquigarrow 0), 1 \vdash^{\mathsf{false}}_{\Sigma,F} \texttt{andlists}(\texttt{l1},\texttt{l3}) : \mathsf{boollist}(0 \rightsquigarrow 0), 1$$

$$\texttt{andlists2} : \mathsf{boollist}(1 \rightsquigarrow 1), \mathsf{boollist}(0 \rightsquigarrow 0), \mathsf{boollist}(0 \rightsquigarrow 0), 1$$
$$\to \mathsf{boollist}(0 \rightsquigarrow 0) \times \mathsf{boollist}(0 \rightsquigarrow 0), 1$$

Another is to reuse the potential on `l1` internally, but assign the final potential to part of the result:

$$\texttt{l1} : \mathsf{boollist}(1 \rightsquigarrow 1), \texttt{l2} : \mathsf{boollist}(0 \rightsquigarrow 0), 1 \vdash^{\mathsf{false}}_{\Sigma,F} \texttt{andlists}(\texttt{l1},\texttt{l2}) : \mathsf{boollist}(0 \rightsquigarrow 0), 1$$
$$\texttt{l1} : \mathsf{boollist}(1 \rightsquigarrow 0), \texttt{l3} : \mathsf{boollist}(0 \rightsquigarrow 0), 1 \vdash^{\mathsf{false}}_{\Sigma,F} \texttt{andlists}(\texttt{l1},\texttt{l3}) : \mathsf{boollist}(1 \rightsquigarrow 0), 1$$

$$\texttt{andlists2} : \mathsf{boollist}(1 \rightsquigarrow 0), \mathsf{boollist}(0 \rightsquigarrow 0), \mathsf{boollist}(0 \rightsquigarrow 0), 1$$
$$\to \mathsf{boollist}(0 \rightsquigarrow 0) \times \mathsf{boollist}(1 \rightsquigarrow 0), 1$$

(In this extension we can only assign the potential to the part of the result evaluated last (`r2`) unless we also increase the overall memory bound. When we consider introducing maximums into the potential functions in later chapters we will be able to assign potential to the first part of the result, too.)

Our final instance is to split the potential between the argument and the result:

$$\texttt{l1} : \mathsf{boollist}(1 \rightsquigarrow 1), \texttt{l2} : \mathsf{boollist}(0 \rightsquigarrow 0), 1 \vdash^{\mathsf{false}}_{\Sigma,F} \texttt{andlists}(\texttt{l1},\texttt{l2}) : \mathsf{boollist}(0 \rightsquigarrow 0), 1$$
$$\texttt{l1} : \mathsf{boollist}(1 \rightsquigarrow \tfrac{1}{2}), \texttt{l3} : \mathsf{boollist}(0 \rightsquigarrow 0), 1 \vdash^{\mathsf{false}}_{\Sigma,F} \texttt{andlists}(\texttt{l1},\texttt{l3}) : \mathsf{boollist}(\tfrac{1}{2} \rightsquigarrow 0), 1$$

$$\texttt{andlists2} : \mathsf{boollist}(1 \rightsquigarrow \tfrac{1}{2}), \mathsf{boollist}(0 \rightsquigarrow 0), \mathsf{boollist}(0 \rightsquigarrow 0), 1$$
$$\rightarrow \mathsf{boollist}(0 \rightsquigarrow 0) \times \mathsf{boollist}(\tfrac{1}{2} \rightsquigarrow 0), 1$$

Notice that the result types also have give-back annotations. These have a slightly different meaning — they indicate that there may be some 'overlap' between the potential assigned to the result and the potential assigned to the arguments. To understand why such overlaps can be useful, consider the following functions:

```
let tail l = match l with cons(h,t)' -> t
let andtail (l1,l2) = let t1 = tail l1 in andlists(t1,l2)
```

One stack frame per element of `l1` is sufficient to evaluate `andtail` and we expect that the potential can be given back to `l1` after evaluation:

$$\texttt{andlists} : \mathsf{boollist}(1 \rightsquigarrow 1), \mathsf{boollist}(0 \rightsquigarrow 0), 0 \rightarrow \mathsf{boollist}(0 \rightsquigarrow 0), 0,$$
$$\texttt{andtail} : \mathsf{boollist}(1 \rightsquigarrow 1), \mathsf{boollist}(0 \rightsquigarrow 0), 1 \rightarrow \mathsf{boollist}(0 \rightsquigarrow 0), 1.$$

What, then, should the signature of `tail` be? It must indicate that `l1` can be reassigned its potential (using the annotations $1 \rightsquigarrow 1$), but only if `t1` can be reassigned its potential after evaluating `andlists` (again, by the type annotations $1 \rightsquigarrow 1$). Thus we give `tail` the signature

$$\texttt{tail} : \mathsf{boollist}(1 \rightsquigarrow 1), 0 \rightarrow \mathsf{boollist}(1 \rightsquigarrow 1), 0,$$

which we take to mean that `l1` can be reassigned 1 unit of potential per element *after* we have finished using `t1`. More precisely, we require that we only reassign the 'given-back' potential to `l1` once we have evaluated an expression whose result does not contain any part of `l1`, because our analysis will ensure that distinct values do not have any overlap in potential.

As `andlists(t1,l2)` returns a freshly allocated list, the result does not contain any part of `l1` and so we can assign the next use of `l1` the same amount of potential it had before. In general we will use the heap separation condition derived from a safety analysis for this (as we noted in Section 2.1).

Major changes to the typing rules from the analysis in Chapter 4 are only required in two places. First, we must introduce a new form of contraction which reassigns the given-back potential where possible. We need to ensure that the two uses of the variable occur sequentially, so we add the new contraction to the LET typing rule, and we will add a side condition to prevent overlapping between those variables and the result of the first subexpression.

Second, we will adjust the pattern matching rules, CASE and CASE′, to 'reserve' the given-back potential in proportion to the data structure's size.

5.2.1 Definition

The annotated types in our new system are

$$T_a := 1 \mid \mathsf{bool} \mid T_a \otimes T_a \mid (T_a, k_l) + (T_a, k_r) \mid ty(\overline{k \rightsquigarrow k'}),$$

where k_l and k_r are constraint variables and $\overline{k \rightsquigarrow k'}$ is a tuple of pairs of constraint variables. This differs from Hofmann-Jost in that all of the annotations in an algebraic datatype $ty(\overline{k \rightsquigarrow k'})$ now come in pairs, $k_i \rightsquigarrow k_i'$. The first annotation k_i in each pair plays the same role as before. The second annotation in each pair, k_i', determines the given-back potential, as outlined above. Function signatures have the same form as before:

$$\Sigma(f) = T_1, \ldots, T_p, k \to T, k' \,|\, \Phi$$

Constructor signatures now have the form

$$\Sigma(c_i) = \forall \overline{k \rightsquigarrow k'}. T_1, \ldots, T_p, k_i \rightsquigarrow k_i' \to ty(\overline{k \rightsquigarrow k'})$$

To avoid cluttering the typing rules with extra constraints we adopt the convention that for any pair of annotations $k \rightsquigarrow k'$ used in the typing we implicitly add the constraint

$$k \geq k'$$

to the linear program. This bounds the give-back annotation k' for unused variables.

When reassigning the given-back potential we need to show that there is no overlap with the potential assigned to the result. As discussed above, we use the heap separation condition from the safety analysis for this. However, some values are not heap allocated and so the separation condition is not useful for them. Thus we do not provide give-back annotations for sum types, and also adopt the convention that for any constructor represented by null (that is, any $c_i \in \mathsf{nullc}$) we add the constraint

$$k_i' = 0$$

to the linear program (where $\Sigma(c_i)[\overline{k \rightsquigarrow k'}] = k_i \rightsquigarrow k_i' \to ty(\overline{k \rightsquigarrow k'})$).

This is a mild restriction because only a fixed amount of potential is involved, which can be represented by the fixed amount of potential in the typing judgements instead.

To calculate the potential prior to evaluation we use the same function as Hofmann-Jost, ignoring the new give-back annotations:

$$\Upsilon : \mathsf{heap} \times \mathsf{val} \times T_a \to \mathbb{Q}^+,$$

$$\Upsilon(\sigma, *, 1) = \Upsilon(\sigma, \mathsf{true}, \mathsf{bool}) = \Upsilon(\sigma, \mathsf{false}, \mathsf{bool}) = 0$$

$$\Upsilon(\sigma, (v', v''), T' \otimes T'') = \Upsilon(\sigma, v', T') + \Upsilon(\sigma, v'', T''),$$

$$\Upsilon(\sigma, \mathsf{inl}(v), (T', k') + (T'', k'')) = k' + \Upsilon(\sigma, v, T'),$$

$$\Upsilon(\sigma, \mathsf{inr}(v), (T', k') + (T'', k'')) = k'' + \Upsilon(\sigma, v, T''),$$

$$\Upsilon(\sigma, \mathsf{null}, ty(\overline{k \rightsquigarrow k'})) = k_i \quad \text{where} \quad c \in \mathsf{nullc}$$
$$\text{and} \quad \Sigma(c)[\overline{k \rightsquigarrow k'}] = k_i \rightsquigarrow k_i' \to ty(\overline{k \rightsquigarrow k'}),$$

$$\Upsilon(\sigma, l, ty(\overline{k \rightsquigarrow k'})) = \sum_{i=1}^{p} \Upsilon(\sigma \setminus l, v_i, T_i) + k_j,$$
$$\text{where} \quad \sigma(l) = (c, v_1, \ldots, v_p),$$
$$\text{and} \quad \Sigma(c)[\overline{k \rightsquigarrow k'}] = T_1, \ldots, T_p, k_j \rightsquigarrow k_j' \to ty(\overline{k \rightsquigarrow k'}).$$

To define the potential after evaluation we also use an extra potential function for the give-back annotations:

$$\Upsilon' : \mathsf{heap} \times \mathsf{val} \times T_a \times \mathsf{loc} \to \mathbb{Q}^+$$

$$\Upsilon'(\sigma, *, 1, l) = \Upsilon'(\sigma, \mathsf{true}, \mathsf{bool}, l) = \Upsilon'(\sigma, \mathsf{false}, \mathsf{bool}, l) = 0$$

$$\Upsilon'(\sigma, (v', v''), T' \otimes T'', l) = \Upsilon'(\sigma, v', T', l) + \Upsilon'(\sigma, v'', T'', l)$$

$$\Upsilon'(\sigma, \mathsf{inl}(v), (T', k') + (T'', k''), l) = \Upsilon'(\sigma, v, T', l)$$

$$\Upsilon'(\sigma, \mathsf{inr}(v), (T', k') + (T'', k''), l) = \Upsilon'(\sigma, v, T'', l)$$

$$\Upsilon'(\sigma, \mathsf{null}, ty(\overline{k \rightsquigarrow k'}), l) = 0$$

$$\Upsilon'(\sigma, l', ty(\overline{k \rightsquigarrow k'}), l) = \sum_{i=1}^{p} \Upsilon'(\sigma \setminus l', v_i, T_i, l) + \begin{cases} k_j' & \text{if } l' = l \\ 0 & \text{otherwise} \end{cases}$$
$$\text{where} \quad \sigma(l) = (c, v_1, \ldots, v_p),$$
$$\text{and} \quad \Sigma(c)[\overline{k \rightsquigarrow k'}] = T_1, \ldots, T_p, k_j \rightsquigarrow k_j' \to ty(\overline{k \rightsquigarrow k'}).$$

In order to eliminate overlaps, Υ' is defined *per heap location* l. The overall post-evaluation potential is the 'normal' potential of the result using Υ, plus the give-back potential less the amount that it overlaps with the result's potential:

$$n' + \Upsilon(\sigma', v, T) + \sum_{l \in \mathsf{loc}} \max\{0, \Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma', v, T, l)\},$$

for a typing judgement $\Gamma, n \vdash^{g,t}_{\Sigma,F} e : T, n' \,|\, \Phi$ and evaluation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$. The $\max\{0, \ldots\}$ part is to prevent freshly allocated parts of the result interfering with

the total potential. We will be able to use the separation condition to eliminate the $\Upsilon'(\sigma', v, T, l)$ term in the soundness proof, removing the overlap.

The typing rules for expressions in the give-back system are presented in Figures 5.1 and 5.2. They are based on the stack and heap analysis from Chapter 4 and only the three highlighted rules, LET-GB, CASE′-GB and CASE-GB, have changed significantly from the earlier system. The remainder of the rules do not involve the new annotations directly, and we only need the extended definition for $\oplus$ in Figure 5.3. The typing rules for functions and programs are presented in Figure 5.4 where the only change is to include the tail position information.

Note that we retain the original contraction rule SHARE because the LET-GB form we have introduced cannot be used in some circumstances; for example, the uses of `l` in `andlists(l,l)` do not occur sequentially, and so we must use SHARE.

Our replacement rule for let expressions, LET-GB, takes advantage of the give-back annotations. For each of the variables appearing in the subcontext Δ, it reassigns the potential represented by the give-back annotation in Δ_1 to the use of the variable in the second subexpression Δ_2. To deal with the overlapping potential issue discussed above we have a side condition requiring that the variables involved in the contraction are separate from the subexpression's result.

We presumed in Section 2.1 the availability of 'benign sharing' analyses that can give a conservative estimate of the set of variables satisfying the separation condition. For instance, Konečný's DEEL typing (Konečný, 2003) can be used. Thus we can use this set during type inference to decide which variables from the context to put into Δ.

The 4-place relation $\cdot = \cdot \succ \cdot \,|\, \cdot$ in Figure 5.5 formalises the use of the give-back annotations. Informally, $\Delta = \Delta_1 \succ \Delta_2 \,|\, \Phi$ means that if the constraints in Φ are satisfied and we are given a context Δ, then we can use Δ_1 when typing one expression, then Δ_2 in a subsequent expression — with Δ_2 using the given-back potential in Δ_1. For example, if we are typing some $\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2$ expression with the context $\Delta = x : \mathsf{boollist}(k \rightsquigarrow k')$ then we have

$$\frac{\Delta = \Delta_1 \succ \Delta_2 \,|\, \Phi_\succ \quad \Delta_1, n \vdash e_1 : T_0, n_0 \,|\, \Phi_1 \quad \Delta_2, y : T_0, n_0 \vdash e_2 : T, n' \,|\, \Phi_2}{\Delta, n \vdash \mathsf{let}\ y = e_1\ \mathsf{in}\ e_2 : T, n' \,|\, \Phi_\succ \cup \Phi_1 \cup \Phi_2}\ \text{LET-GB}$$

assuming that the result of e_1 is separate from x, and $\Delta_i = x : \mathsf{boollist}(k_i \rightsquigarrow k_i')$. The judgement for $\succ$ yields the constraints

$$\Phi_\succ = \{k \geq k_1, k - k_1 + k_1' \geq k_2, k_2' \geq k'\}$$

which mean that when using x in e_1, the potential corresponding to k_1' out of the k_1

$$\frac{}{\Gamma, n \vdash^{g,t}_{\Sigma,F} * : 1, n' \,|\, \{n \geq n'\}} \text{ (UNIT)} \qquad \frac{c \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma, n \vdash^{g,t}_{\Sigma,F} c : \mathsf{bool}, n' \,|\, \{n \geq n'\}} \text{ (BOOL)}$$

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma, n \vdash^{g,t}_{\Sigma,F} x : \Gamma(x), n' \,|\, \{n \geq n'\}} \text{ (VAR)}$$

$$\frac{\begin{array}{c} f \in F \qquad \Sigma(f) = T'_1, \ldots, T'_p, k \to T', k' | \Phi' \qquad \rho(T'_i) = T_i \qquad \rho(T') = T \\ \Phi = \rho(\Phi') \cup \{n \geq \rho(k) + \mathrm{stack}'(g, f, t), n - \rho(k) + \rho(k') \geq n'\} \end{array}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash^{g,t}_{\Sigma,F} f(x_1, \ldots, x_p) : T, n' \,|\, \Phi} \text{ (FUN-TAIL)}$$

$$\frac{\begin{array}{c} f \notin F \qquad \Sigma(f) = T_1, \ldots, T_p, k \to T, k' | \Phi' \\ \Phi = \{n \geq k + \mathrm{stack}'(g, f, t), n - k + k' \geq n'\} \end{array}}{\Gamma, x_1 : T_1, \ldots, x_p : T_p, n \vdash^{g,t}_{\Sigma,F} f(x_1, \ldots, x_p) : T, n' \,|\, \Phi} \text{ (FUNDEF-TAIL)}$$

$$\frac{\begin{array}{c} \Gamma_1, \Delta_1, n \vdash^{g,\mathsf{false}}_{\Sigma,F} e_1 : T_0, n_0 \,|\, \Phi_1 \qquad \Gamma_2, \Delta_2, x : T_0, n_0 \vdash^{g,t}_{\Sigma,F} e_2 : T, n' \,|\, \Phi_2 \\ \Delta = \Delta_1 \succ \Delta_2 \,|\, \Phi_3 \qquad \text{Values for } \Delta \text{ are separate from the result of } e_1 \end{array}}{\Gamma_1, \Gamma_2, \Delta, n \vdash^{g,t}_{\Sigma,F} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T, n' \,|\, \Phi_1 \cup \Phi_2 \cup \Phi_3} \text{ (LET-GB)}$$

$$\frac{\Gamma, n \vdash^{g,t}_{\Sigma,F} e_t : T, n' \,|\, \Phi_1 \qquad \Gamma, n \vdash^{g,t}_{\Sigma,F} e_f : T, n' \,|\, \Phi_2}{\Gamma, x : \mathsf{bool}, n \vdash^{g,t}_{\Sigma,F} \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : T, n' \,|\, \Phi_1 \cup \Phi_2} \text{ (IF)}$$

$$\frac{}{\Gamma, x_1 : T_1, x_2 : T_2, n \vdash^{g,t}_{\Sigma,F} (x_1, x_2) : T_1 \otimes T_2, n' \,|\, \{n \geq n'\}} \text{ (PAIR)}$$

$$\frac{\Gamma, x_1 : T_1, x_2 : T_2, n \vdash^{g,t}_{\Sigma,F} e : T, n' \,|\, \Phi}{\Gamma, x : T_1 \otimes T_2, n \vdash^{g,t}_{\Sigma,F} \mathsf{match}\ x\ \mathsf{with}\ (x_1, x_2) \to e : T, n' \,|\, \Phi} \text{ (PAIRELIM)}$$

$$\frac{}{\Gamma, x : T_l, n \vdash^{g,t}_{\Sigma,F} \mathsf{inl}(x) : (T_l, k_l) + (T_r, k_r), n' \,|\, \{n \geq k_l + n'\}} \text{ (INL)}$$

$$\frac{}{\Gamma, x : T_r, n \vdash^{g,t}_{\Sigma,F} \mathsf{inr}(x) : (T_l, k_l) + (T_r, k_r), n' \,|\, \{n \geq k_r + n'\}} \text{ (INR)}$$

$$\frac{\begin{array}{c} \Gamma, x_l : T_l, n_l \vdash^{g,t}_{\Sigma,F} e_l : T, n' \,|\, \Phi_l \qquad \Gamma, x_r : T_r, n_r \vdash^{g,t}_{\Sigma,F} e_r : T, n' \,|\, \Phi_r \\ \Phi = \Phi_l \cup \Phi_r \cup \{n_l = n + k_l, n_r = n + k_r\} \end{array}}{\Gamma, x : (T_l, k_l) + (T_r, k_r), n \vdash^{g,t}_{\Sigma,F} \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r : T, n' \,|\, \Phi} \text{ (SUMELIM)}$$

$$\frac{\Gamma, a : T_1, b : T_2, n \vdash^{g,t}_{\Sigma,F} e : T', n' \,|\, \Phi \qquad T = T_1 \oplus T_2 \,|\, \Phi'}{\Gamma, x : T, n \vdash^{g,t}_{\Sigma,F} e[x/a, x/b] : T', n' \,|\, \Phi \cup \Phi'} \text{ (SHARE)}$$

Figure 5.1: Typing rules for expressions in the give-back analysis

$$\frac{\begin{array}{c}\Sigma(c_i)[\overline{k \rightsquigarrow k'}] = T_1,\ldots,T_p,k_i \rightsquigarrow k_i' \to ty(\overline{k \rightsquigarrow k'}) \\ \Phi = \{n \geq \mathrm{size}(c_i) + k_i + n'\}\end{array}}{\Gamma, x_1 : T_1,\ldots,x_p : T_p, n \vdash^{g,t}_{\Sigma,F} c_i(x_1,\ldots,x_p) : ty(\overline{k \rightsquigarrow k'}), n' \,|\, \Phi} \text{ (CONSTRUCT)}$$

$$\frac{\text{for all } i,\ 1 \leq i \leq m, \quad \Gamma, n \triangleright ty(\overline{k \rightsquigarrow k'}) \vdash^{g,t}_{\Sigma,F} p_i \to e_i : T', n' \,|\, \Phi_i}{\Gamma, x : ty(\overline{k \rightsquigarrow k'}), n \vdash^{g,t}_{\Sigma,F} \mathsf{match}\ x\ \mathsf{with}\ p_1 \to e_1 \triangleright \cdots \triangleright p_m \to e_m : T', n' \,|\, \bigcup_i \Phi_i} \text{ (MATCH)}$$

$$\frac{\begin{array}{c}\Gamma, x_1 : T_1,\ldots,x_p : T_p, n_i \vdash^{g,t}_{\Sigma,F} e : T', n_i' \,|\, \Phi \\ \Phi' = \{n_i = n + k_i + \mathrm{size}(c_i), n_i' = n' + k_i'\} \\ \Sigma(c_i)[\overline{k \rightsquigarrow k'}] = T_1,\ldots,T_p,k_i \rightsquigarrow k_i' \to ty(\overline{k \rightsquigarrow k'})\end{array}}{\Gamma, n \triangleright ty(\overline{k \rightsquigarrow k'}) \vdash^{g,t}_{\Sigma,F} c_i(x_1,\ldots,x_p) \to e : T', n' \,|\, \Phi \cup \Phi'} \text{ (CASE-GB)}$$

$$\frac{\begin{array}{c}\Gamma, x_1 : T_1,\ldots,x_p : T_p, n_i \vdash^{g,t}_{\Sigma,F} e : T', n_i' \,|\, \Phi \\ \Phi' = \{n_i = n + k_i, n_i' = n' + k_i'\} \\ \Sigma(c_i)[\overline{k \rightsquigarrow k'}] = T_1,\ldots,T_p,k_i \rightsquigarrow k_i' \to ty(\overline{k \rightsquigarrow k'})\end{array}}{\Gamma, n \triangleright ty(\overline{k \rightsquigarrow k'}) \vdash^{g,t}_{\Sigma,F} c_i(x_1,\ldots,x_p)' \to e : T', n' \,|\, \Phi \cup \Phi'} \text{ (CASE'-GB)}$$

Figure 5.2: Typing rules for expressions in the give-back analysis (continued)

$$\frac{}{1 = 1 \oplus 1 \,|\, \emptyset} \qquad \frac{}{\mathsf{bool} = \mathsf{bool} \oplus \mathsf{bool} \,|\, \emptyset}$$

$$\frac{T = T_1 \oplus T_2 \,|\, \Phi \qquad T' = T_1' \oplus T_2' \,|\, \Phi'}{T \otimes T' = (T_1 \otimes T_1') \oplus (T_2 \otimes T_2') \,|\, \Phi \cup \Phi'}$$

$$\frac{T = T_1 \oplus T_2 \,|\, \Phi \qquad T' = T_1' \oplus T_2' \,|\, \Phi' \qquad \Phi'' = \{k = k_1 + k_2, k' = k_1' + k_2'\}}{(T,k) + (T',k') = (T_1,k_1) + (T_1',k_1') \oplus (T_2,k_2) + (T_2',k_2') \,|\, \Phi \cup \Phi' \cup \Phi''}$$

$$\frac{}{ty(\overline{k \rightsquigarrow k'}) = ty(\overline{k_1 \rightsquigarrow k_1'}) \oplus ty(\overline{k_2 \rightsquigarrow k_2'}) \mid \{k_i = k_{1,i} + k_{2,i}, k_i' = k_{1,i}' + k_{2,i}' : \forall i\}}$$

Figure 5.3: Rules for splitting with give-back annotations

$$\frac{\Sigma(f) = T_1,\ldots,T_p,k \to T,k' | \Phi \qquad x_1 : T_1,\ldots,x_p : T_p, k \vdash^{f,\mathsf{true}}_{\Sigma,F} e_f : T,k' \,|\, \Phi'}{\vdash_{\Sigma,F} f(x_1,\ldots,x_p) = e_f \Rightarrow \{f\}, \Phi'}$$

$$\frac{\vdash_{\Sigma,F} D \Rightarrow F', \Phi' \qquad \vdash_{\Sigma,F} B \Rightarrow F'', \Phi''}{\vdash_{\Sigma,F} D \text{ and } B \Rightarrow F' \cup F'', \Phi' \cup \Phi''}$$

$$\frac{\vdash_{\Sigma,F} B \Rightarrow F', \Phi' \quad \forall f \in F'. \quad \Sigma(f) = \ldots | \Phi'}{\vdash_{\Sigma,F} \mathsf{let}\ B} \qquad \frac{\vdash_{\Sigma,F} B \Rightarrow F', \Phi' \qquad \vdash_{\Sigma,F \cup F'} P \quad \forall f \in F'. \quad \Sigma(f) = \ldots | \Phi'}{\vdash_{\Sigma,F} \mathsf{let}\ B\ P}$$

Figure 5.4: Typing rules for function signatures in the give-back analysis

$$\frac{}{1 = 1 \succ 1 \,|\, \emptyset} \qquad \frac{}{\mathsf{bool} = \mathsf{bool} \succ \mathsf{bool} \,|\, \emptyset}$$

$$\frac{T = T_1 \succ T_2 \,|\, \Phi \qquad T' = T_1' \succ T_2' \,|\, \Phi'}{T \otimes T' = (T_1 \otimes T_1') \succ (T_2 \otimes T_2') \,|\, \Phi \cup \Phi'}$$

$$\frac{T = T_1 \succ T_2 \,|\, \Phi \qquad T' = T_1' \succ T_2' \,|\, \Phi' \qquad \Phi'' = \{k = k_1 + k_2, k' = k_1' + k_2'\}}{(T,k) + (T',k') = (T_1,k_1) + (T_1',k_1') \succ (T_2,k_2) + (T_2',k_2') \,|\, \Phi \cup \Phi' \cup \Phi''}$$

$$\frac{\Phi = \{k_{0,i} \geq k_{1,i},\ k_{0,i} - k_{1,i} + k_{1,i}' \geq k_{2,i},\ k_{2,i}' \geq k_{0,i}' : \forall i\}}{ty(\overline{k_0 \rightsquigarrow k_0'}) = ty(\overline{k_1 \rightsquigarrow k_1'}) \succ ty(\overline{k_2 \rightsquigarrow k_2'}) \mid \Phi}$$

$$\frac{\forall x \in \mathrm{dom}(\Delta).\ \Delta(x) = \Delta_1(x) \succ \Delta_2(x) \,|\, \Phi_x}{\Delta = \Delta_1 \succ \Delta_2 \,|\, \cup_{x \in \mathrm{dom}(\Delta)} \Phi_x}$$

Figure 5.5: Rules for contraction involving the give-back annotations

annotation in Δ_1 is only needed temporarily, so it can be reused during the execution of e_2 through the k_2 annotation in Δ_2. (The portion of k that is not used in e_1, $k - k_1$, can also be added to k_2.)

Finally, the pattern matching rules enforce the existence of the potential represented by the give-back annotations. Consider its effect when matching a $\mathsf{boollist}(k \rightsquigarrow k')$. The CASE′ rule adds the 'normal' list annotation k to the fixed annotation n when typing the cons case because we are removing an element from the list, and so get one element's worth of potential to use in the subexpression. Our replacement rule, CASE′-GB, also requires the potential for the 'give-back' annotation to be returned afterwards for a later use of x. We adapt CASE in the same way.

5.2.2 Soundness

To show that the heap and stack memory bounds derived from the potential are sufficient for evaluation we will prove that the typing and corresponding evaluation rules preserve the invariant that the free memory is greater than the potential throughout the program's execution. In order to perform the induction we will also show that any excess free memory is preserved[1].

Before the main theorem we will establish several lemmas about the behaviour of the potential functions. First, we show that Υ and Υ' are linear with respect to the type

[1]This is unsurprising because LFD programs have no way to adjust their behaviour to take advantage of extra free memory. Indeed, they do not even have any way to observe the amount of free memory.

operator $\oplus$ used in the SHARE rule.

Lemma 5.3. *For any σ, v, T, T_1, T_2 and l, if we have*

$$T = T_1 \oplus T_2 \,|\, \Phi$$

then for any solution of Φ

$$\begin{aligned}\Upsilon(\sigma, v, T) =& \Upsilon(\sigma, v, T_1) + \Upsilon(\sigma, v, T_2), \text{ and} \\ \Upsilon'(\sigma, v, T, l) =& \Upsilon'(\sigma, v, T_1, l) + \Upsilon'(\sigma, v, T_2, l).\end{aligned}$$

Proof. For the first equation, we use induction on the size of σ, v. The unit and boolean cases are trivial. For pairs, $v = (v', v'')$ and $T = T' \otimes T''$. From the definition of $\oplus$ there exist T_1', T_1'', T_2', T_2'' such that $T_1 = T_1' \otimes T_1''$, $T_2 = T_2' \otimes T_2''$ and

$$T' = T_1' \oplus T_2' \,|\, \Phi' \quad T'' = T_1'' \oplus T_2'' \,|\, \Phi''$$

for some $\Phi', \Phi'' \subseteq \Phi$. So using the definition of Υ and the induction hypothesis we have

$$\begin{aligned}\Upsilon(\sigma, v, T) &= \Upsilon(\sigma, (v', v''), T' \otimes T'') \\ &= \Upsilon(\sigma, v', T') + \Upsilon(\sigma, v'', T'') \\ &= \Upsilon(\sigma, v', T_1') + \Upsilon(\sigma, v', T_2') + \Upsilon(\sigma, v'', T_1'') + \Upsilon(\sigma, v'', T_2'') \\ &= \Upsilon(\sigma, v, T_1) + \Upsilon(\sigma, v, T_2).\end{aligned}$$

For sums, suppose that $v = \mathsf{inl}(v')$ (the inr case is similar) and $T = (T', k') + (T'', k'')$. From $\oplus$ we have T_i', T_i'', k_i', k_i'' such that $T_1 = (T_1', k_1') + (T_1'', k_1'')$ and similarly for T_2, with $k' = k_1' + k_2'$ and $k'' = k_1'' + k_2''$. Also,

$$T' = T_1' \oplus T_2' \,|\, \Phi' \quad T'' = T_1'' \oplus T_2'' \,|\, \Phi''$$

for some $\Phi', \Phi'' \subseteq \Phi$. Again, from the definition of Υ and the induction hypothesis we have

$$\begin{aligned}\Upsilon(\sigma, v, T) &= \Upsilon(\sigma, \mathsf{inl}(v'), (T', k') + (T'', k'')) \\ &= k' + \Upsilon(\sigma, v', T') \\ &= k_1' + k_2' + \Upsilon(\sigma, v', T') \\ &= k_1' + k_2' + \Upsilon(\sigma, v', T_1') + \Upsilon(\sigma, v', T_2') \\ &= \Upsilon(\sigma, v, T_1) + \Upsilon(\sigma, v, T_2).\end{aligned}$$

Finally, for datatype values $v = l'$ for some $l' \in \mathsf{loc}$, and $T = ty(\overline{k \rightsquigarrow k'})$. From $\oplus$ we have $T_1 = ty(\overline{k_1 \rightsquigarrow k'_1})$ and $T_2 = ty(\overline{k_2 \rightsquigarrow k'_2})$ with $k_j = k_{1,j} + k_{2,j}$ for all j. If $l' = \mathsf{null}$ then there is a unique $c \in \mathsf{nullc}$ for type ty and $\Sigma(c)[\overline{k \rightsquigarrow k'}] = k_i \rightsquigarrow k'_i \to ty(\overline{k \rightsquigarrow k'})$. Then

$$\Upsilon(\sigma, v, T) = k_i = k_{1,i} + k_{2,i} = \Upsilon(\sigma, v, T_1) + \Upsilon(\sigma, v, T_2).$$

If $l' \neq \mathsf{null}$ then $\sigma(l')$ has the form $(c, v_1, \ldots, v_p)$ with some signature $\Sigma(c)[\overline{k \rightsquigarrow k'}] = T_1, \ldots, T_p, k_i \rightsquigarrow k'_i \to ty(\overline{k \rightsquigarrow k'})$. Hence

$$\begin{aligned}
\Upsilon(\sigma, v, T) &= \sum_{j=1}^{p} \Upsilon(\sigma \setminus l', v_j, T_j) + k_i \\
&= \sum_{j=1}^{p} (\Upsilon(\sigma \setminus l', v_j, T_{1,j}) + \Upsilon(\sigma \setminus l', v_j, T_{2,j})) + k_{1,i} + k_{2,i} \\
&= \Upsilon(\sigma, v, T_1) + \Upsilon(\sigma, v, T_2).
\end{aligned}$$

using the induction hypothesis for the second step.

The second part takes the same form as the first, except that we omit the constant in the $T = (T', k') + (T'', k'')$ case and for the algebraic datatypes case we use $k'_i = k'_{1,i} + k'_{2,i}$ as follows:

$$\begin{aligned}
\Upsilon'(\sigma, v, T, l) &= \sum_{j=1}^{p} \Upsilon'(\sigma \setminus l', v_j, T_j, l) + \begin{cases} k'_i & \text{if } l' = l \\ 0 & \text{otherwise} \end{cases} \\
&= \sum_{j=1}^{p} (\Upsilon'(\sigma \setminus l', v_j, T_{1,j}, l) + \Upsilon'(\sigma \setminus l', v_j, T_{2,j}, l)) + \begin{cases} k'_{1,i} + k'_{2,i} & \text{if } l' = l \\ 0 & \text{otherwise} \end{cases} \\
&= \Upsilon'(\sigma, v, T_1, l) + \Upsilon'(\sigma, v, T_2, l).
\end{aligned}$$

□

The corresponding result for the $\succ$ relation establishes the potential for each variable in Δ in each subexpression:

Lemma 5.4. *Given $T = T_1 \succ T_2 \,|\, \Phi$, we have*

$$\begin{gathered}
\Upsilon(\sigma, v, T) \geq \Upsilon(\sigma, v, T_1), \\
\Upsilon'(\sigma, v, T_2, l) \geq \Upsilon'(\sigma, v, T, l) \quad \textit{and} \\
\sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, v, T_1, l) + \Upsilon(\sigma, v, T) - \Upsilon(\sigma, v, T_1) \geq \Upsilon(\sigma, v, T_2).
\end{gathered}$$

Proof. The first two equations follow from the definitions. The $\cdot = \cdot \succ \cdot \,|\, \cdot$ relation ensures that the 'normal' annotations in T_1 are no larger than the corresponding annotations in T. So a simple induction on Υ shows that $\Upsilon(\sigma, v, T) \geq \Upsilon(\sigma, v, T_1)$. The second equation is similar.

For the last equation we use induction on the size of σ, v. Unit and boolean values are trivial. Pairs reduce immediately to the induction hypothesis. For sums, suppose that $v = \mathsf{inl}(v')$ for some v' and $T_i = (T_i', k_i') + (T_i'', k_i'')$. Then

$$\begin{aligned}
&\sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, v, T_1, l) + \Upsilon(\sigma, v, T) - \Upsilon(\sigma, v, T_1) \\
&\qquad = \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, v', T_1', l) + \big(k + \Upsilon(\sigma, v', T')\big) - (k_1 + \Upsilon(\sigma, v', T_1')) \\
&\qquad \geq k_2 + \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, v', T_1', l) + \Upsilon(\sigma, v', T') - \Upsilon(\sigma, v', T_1') \\
&\qquad \geq k_2 + \Upsilon(\sigma, v', T_2') \\
&\qquad = \Upsilon(\sigma, v, T_2).
\end{aligned}$$

The inr case is similar.

Finally, for algebraic datatypes we have $v = l' \in \mathsf{loc}, T = ty(\overline{k_0 \rightsquigarrow k_0'})$. The definition of the $\succ$ relation gives us $T_1 = ty(\overline{k_1 \rightsquigarrow k_1'})$ and $T_2 = ty(\overline{k_2 \rightsquigarrow k_2'})$ with $k_{0,i} - k_{1,i} + k_{1,i}' \geq k_{2,i}$. If $l' = \mathsf{null}$ then

$$\sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, l', T_1, l) + \Upsilon(\sigma, l', T) - \Upsilon(\sigma, l', T_1) = k_{1,i}' + k_{0,i} - k_{1,i} \geq k_{2,i} = \Upsilon(\sigma, v, T_2),$$

because $\Upsilon'(\sigma, l', T_1, l) = 0$ by definition, and $k_{1,i}' = 0$ by our convention on give-back annotations for constructors represented by null.

If $l \neq \mathsf{null}$ then $\sigma(l') = (c, v_1, \ldots, v_p)$ with $\Sigma(c)[\overline{k_0 \rightsquigarrow k_0'}] = T_1, \ldots, T_p, k_{0,i} \rightsquigarrow k_{0,i}' \to ty(\overline{k_0 \rightsquigarrow k_0'})$. Thus,

$$\begin{aligned}
&\sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, v, T_1, l) + \Upsilon(\sigma, l', T) - \Upsilon(\sigma, l', T_1) \\
&\qquad = k_{1,i}' + \sum_{l \in \mathsf{loc}} \sum_{j=1}^{p} \Upsilon'(\sigma \setminus l', v_j, T_{1,j}, l) \\
&\qquad\qquad + k_{0,i} + \sum_{j=1}^{p} \Upsilon(\sigma \setminus l', v_j, T_j) - \left(k_{1,i} + \sum_{j=1}^{p} \Upsilon(\sigma \setminus l', v_j, T_{1,j}) \right) \\
&\qquad \geq k_{0,i} - k_{1,i} + k_{1,i}' \\
&\qquad\qquad + \sum_{j=1}^{p} \left(\sum_{l \in \mathsf{loc}} \Upsilon'(\sigma \setminus l', v_j, T_{1,j}, l) + \Upsilon(\sigma \setminus l', v_j, T_j) - \Upsilon(\sigma \setminus l', v_j, T_{1,j}) \right) \\
&\qquad \geq k_{2,i} + \sum_{j=1}^{p} \Upsilon(\sigma \setminus l', v_j, T_{2,j}) \\
&\qquad = \Upsilon(\sigma, l', T_2),
\end{aligned}$$

as required. □

We must also show that Υ gives a bound on Υ' to deal with unused parts of the context:

Lemma 5.5. *For any σ, v, T, if $\Upsilon(\sigma, v, T)$ is defined then*

$$\Upsilon(\sigma, v, T) \geq \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, v, T, l).$$

Proof. As before, the proof uses induction on the size of σ, v. In each case we expand the definition of Υ, use the induction hypothesis and finish using the definition of Υ'. We present the non-null datatype value case as it is the most involved.

Suppose $v = l' \in \mathsf{loc}$, $\sigma(l') = (c, v_1, \ldots, v_p)$ and $T = ty(\overline{k \rightsquigarrow k'})$ with $\Sigma(c)[\overline{k \rightsquigarrow k'}] = T_1, \ldots, T_p, k_i \rightsquigarrow k_i' \to T$. By our convention on annotations, $k_i \geq k_i'$. So, using the definitions and the induction hypothesis,

$$\begin{aligned}
\Upsilon(\sigma, v, T) &= \sum_{j=1}^{p} \Upsilon(\sigma \setminus l', v_j, T_j) + k_i \\
&\geq \sum_{j=1}^{p} \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma \setminus l', v_j, T_j, l) + k_i' \\
&= \sum_{l \in \mathsf{loc}} \left(\sum_{j=1}^{p} \Upsilon'(\sigma \setminus l', v_j, T_j, l) + \left\{ \begin{array}{ll} k_i' & \text{if } l' = l \\ 0 & \text{otherwise} \end{array} \right. \right) \\
&= \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, v, T, l).
\end{aligned}$$

□

We can lift these results to environments and typing contexts:

Corollary 5.6. *Given $\Gamma = \Gamma_1 \oplus \Gamma_2 \,|\, \Phi$ then for any solution of Φ we have*

$$\Upsilon(\sigma, S, \Gamma) = \Upsilon(\sigma, S, \Gamma_1) + \Upsilon(\sigma, S, \Gamma_2), \tag{5.1}$$

$$\Upsilon'(\sigma, S, \Gamma, l) = \Upsilon'(\sigma, S, \Gamma_1, l) + \Upsilon'(\sigma, S, \Gamma_2, l), \tag{5.2}$$

whenever the left hand side is defined. Also, for any value v and type T,

$$\Upsilon(\sigma, S[x \mapsto v], (\Gamma, x : T)) = \Upsilon(\sigma, S, \Gamma) + \Upsilon(\sigma, v, T), \tag{5.3}$$

$$\Upsilon'(\sigma, S[x \mapsto v], (\Gamma, x : T), l) = \Upsilon'(\sigma, S, \Gamma, l) + \Upsilon'(\sigma, v, T, l). \tag{5.4}$$

Given $\Gamma = \Gamma_1 \succ \Gamma_2 \,|\, \Phi$ then for any solution of Φ we have

$$\Upsilon(\sigma, S, \Gamma) \geq \Upsilon(\sigma, S, \Gamma_1), \tag{5.5}$$

$$\Upsilon'(\sigma, S, \Gamma_2, l) \geq \Upsilon'(\sigma, S, \Gamma, l), \tag{5.6}$$

$$\sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, S, \Gamma_1, l) + \Upsilon(\sigma, S, \Gamma) - \Upsilon(\sigma, S, \Gamma_1) \geq \Upsilon(\sigma, S, \Gamma_2), \tag{5.7}$$

whenever $\Upsilon(\sigma, S, \Gamma)$ *is defined. Finally,*

$$\Upsilon(\sigma, S, \Gamma) \geq \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, S, \Gamma, l). \tag{5.8}$$

Proof. Both Υ and Υ' are defined on environments and typing contexts as sums over the contents of the context. All of the properties given are preserved over sums. □

We also need to know that bounds are only affected by relevant parts of the state to allow for changes caused by the allocation and deallocation of other data structures:

Lemma 5.7. *For any* σ, S, Γ*, if* $\Upsilon(\sigma, S, \Gamma)$ *is defined then*

$$\Upsilon(\sigma, S, \Gamma) = \Upsilon(\sigma \upharpoonright \mathcal{R}(\sigma, S'), S', \Gamma).$$

where $S' = S \upharpoonright \mathrm{dom}(\Gamma)$*. Similarly for* Υ'*.*

Proof. Straightforward induction on the values and state; Υ and Υ' use exactly the locations in σ which are reachable from S'. □

Similarly, we also need to be able to weaken away unnecessary parts of the context because the benign sharing conditions only apply to the free variables of the expression:

Lemma 5.8. *Given a derivation of*

$$\Gamma, n \vdash^{g,t}_{\Sigma,F} e : T, n' \mid \Phi$$

then there exists a derivation of

$$(\Gamma \upharpoonright \mathrm{FV}(e)), n \vdash^{g,t}_{\Sigma,F} e : T, n' \mid \Phi'$$

for some $\Phi' \subseteq \Phi$*.*

Proof. A straightforward induction on the derivation. The only rules which can affect a variable that is not in $\mathrm{FV}(e)$ are for contraction (SHARE and LET-GB). These rules merely add extra constraints on the type annotations for such variables. Thus we can construct a typing without them. □

We can now prove the main soundness result — that any assignment of potential which satisfies the constraints from a typing provides an upper bound on the free memory required to evaluate an expression and a lower bound on the free memory afterwards. Any extra potential, q, is preserved.

Theorem 5.9. *If an expression e in a well-typed program has a typing*

$$\Gamma, n \vdash^{g,t}_{\Sigma,F} e : T, n' \,|\, \Phi$$

with an assignment of nonnegative rationals to constraint variables which satisfies the constraints in g's signature, and an evaluation $S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma'$ which satisfies the benign sharing conditions, and $\Upsilon(\sigma, S, \Gamma)$ is defined, then for any $q \in \mathbb{Q}^+$ and $m \in \mathbb{N}$ such that

$$m \geq n + \Upsilon(\sigma, S, \Gamma) + q$$

we have $m, S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma', m'$ where

$$m' \geq n' + \sum_{l \in \mathsf{loc}} \max\{0, \Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma', v, T, l)\} + \Upsilon(\sigma', v, T) + q.$$

Recall that the give-back potential is given as $\sum_{l \in \mathsf{loc}} \max\{0, \dots\}$ to subtract the overlap between the give-back potential of the context and the give-back potential of the result; the max is present so that we do not subtract potential for freshly constructed parts of the result.

Proof. We proceed by simultaneous induction on the evaluation and typing derivations. The evaluation terminates, so the derivation of the evaluation must be finite. The SHARE rule is the only one which has no counterpart in the operational semantics, so we consider it separately.

Note that to obtain $m, S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma', m'$ we only need to show that m is sufficiently large because we already have $S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma'$. We will use Equation 5.8 in the leaf rules UNIT, BOOL, VAR, PAIR, INL, INR, CONSTRUCT, FUN-TAIL and FUNDEF-TAIL to deal with unused parts of the context.

SHARE. We have $\Gamma = \Gamma_0, x : T$ and $S = S_0[x \mapsto v_x]$ for some Γ_0 and S_0 where $v_x = S(x)$, so

$$\begin{aligned} m \geq n + \Upsilon(\sigma, S, \Gamma) + q &= n + \Upsilon(\sigma, S_0, \Gamma_0) + \Upsilon(\sigma, v_x, T) + q \\ &= n + \Upsilon(\sigma, S_0, \Gamma_0) + \Upsilon(\sigma, v_x, T_1) + \Upsilon(\sigma, v_x, T_2) + q \\ &= n + \Upsilon(\sigma, S_0[a \mapsto v_x, b \mapsto v_x], (\Gamma_0, a : T_1, b : T_2)) + q \end{aligned}$$

by the linearity of Υ (Equation 5.1) and context extension (Equation 5.3). Thus by substitution of a and b for x in the appropriate parts of the execution derivation, we can apply the induction hypothesis and obtain the result by the corresponding results for Υ' (Equations 5.2 and 5.4).

UNIT, BOOL. From the definition, $\Upsilon(\sigma, v, T) = \Upsilon'(\sigma, v, T, l) = 0$, so

$$\begin{aligned} m' = m &\geq n + \Upsilon(\sigma, S, \Gamma) + q \\ &\geq n' + \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, S, \Gamma, l) + q \\ &\geq n' + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma, v, T, l)\right\} + \Upsilon(\sigma, v, T) + q. \end{aligned}$$

VAR. The context extension property (Equation 5.3) allows us to extract a variable's contribution to the bound:

$$\begin{aligned} m' = m &\geq n + \Upsilon(\sigma, S, \Gamma) + q \\ &= n + \Upsilon(\sigma, S, \Gamma \setminus x) + \Upsilon(\sigma, S(x), \Gamma(x)) + q \\ &\geq n' + \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, S, \Gamma \setminus x, l) + \Upsilon(\sigma, S(x), \Gamma(x)) + q \\ &= n' + \sum_{l \in \mathsf{loc}} \left(\Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma, S(x), \Gamma(x), l)\right) + \Upsilon(\sigma, S(x), \Gamma(x)) + q \\ &= n' + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma, S(x), \Gamma(x), l)\right\} + \Upsilon(\sigma, S(x), \Gamma(x)) + q. \end{aligned}$$

PAIR. Similar to VAR, except that we extract two variables and use

$$\Upsilon(\sigma, v', T') + \Upsilon(\sigma, v'', T'') = \Upsilon(\sigma, (v', v''), T' \otimes T'')$$

from the definition of Υ to check that $m' = m$ is sufficient.

INL (and INR). Similar to VAR, except that we use the constraint $n \geq k_l + n'$ from the typing rule to allow the use of

$$\Upsilon(\sigma, S(x), T_l) + k_l = \Upsilon(\sigma, \mathsf{inl}(S(x)), (T_l, k_l) + (T_r, k_r))$$

from the definition of Υ when checking that $m' = m$ is sufficient. INR follows in the same manner.

CONSTRUCT with E-CONSTRUCTN. The value $v = \mathsf{null}$. Using the constraint on n from the typing rule to provide the k_i for $\Upsilon(\sigma, \mathsf{null}, ty(\overline{k \rightsquigarrow k'}))$,

$$\begin{aligned} m &\geq n + \Upsilon(\sigma, S, \Gamma) + q \\ &\geq k_i + n' + \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, S, \Gamma, l) + q \\ &\geq n' + \sum_{l \in \mathsf{loc}} \max\{0, \Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma, \mathsf{null}, ty(\overline{k \rightsquigarrow k'}), l)\} + \Upsilon(\sigma, \mathsf{null}, ty(\overline{k \rightsquigarrow k'})) + q, \end{aligned}$$

because $\Upsilon'(\sigma, \mathsf{null}, ty(\overline{k \rightsquigarrow k'}), l) = 0$ by definition.

CONSTRUCT with E-CONSTRUCT. First we note that the constraint on n in the typing rules is large enough for E-CONSTRUCT, which requires at least $\text{size}(c)$ units of memory:

$$m \geq n + \Upsilon(\sigma, S, \Gamma) + q \geq \text{size}(c) + k_i + n' + \Upsilon(\sigma, S, \Gamma) + q$$

The location for the newly created cell is fresh, that is $l \notin \text{dom}(\sigma)$. Thus extending σ to $\sigma' = \sigma[l \mapsto (c, S(x_1), \ldots, S(x_p))]$ preserves the bound $\Upsilon(\sigma, S(x), \Gamma(x)) = \Upsilon(\sigma', S(x), \Gamma(x))$ for any variable $x \in \text{dom}(\Gamma)$. Hence,

$$\begin{aligned}
m' &= m - \text{size}(c) \\
&\geq n' + k_i + \Upsilon(\sigma, S, \Gamma) + q \\
&\geq n' + k_i + \Upsilon(\sigma, S, \Gamma \setminus \{x_1, \ldots, x_p\}) + \sum_{j=1}^{p} \Upsilon(\sigma, S(x_j), T_j) + q \\
&\geq n' + \Upsilon(\sigma, S, \Gamma \setminus \{x_1, \ldots, x_p\}) + \Upsilon(\sigma', l, ty(\overline{k \rightsquigarrow k'})) + q \\
&\geq n' + \sum_{l' \in \mathsf{loc}} \Upsilon'(\sigma, S, \Gamma \setminus \{x_1, \ldots, x_p\}, l') + \Upsilon(\sigma', l, ty(\overline{k \rightsquigarrow k'})) + q \\
&\geq n' + \sum_{l' \in \mathsf{loc}} \max\{0, \Upsilon'(\sigma, S, \Gamma, l') - \Upsilon'(\sigma', l, ty(\overline{k \rightsquigarrow k'}), l')\} + \Upsilon(\sigma', l, ty(\overline{k \rightsquigarrow k'})) + q
\end{aligned}$$

as required.

LET-GB. We can show that m is large enough to apply the induction hypothesis to e_1, while incorporating the remaining amount into the constant q so that we can use it for e_2:

$$\begin{aligned}
m &\geq n + \Upsilon(\sigma, S, (\Gamma_1, \Gamma_2, \Delta)) + q \\
&\geq n + \Upsilon(\sigma, S, (\Gamma_1, \Delta_1)) + (\Upsilon(\sigma, S, (\Gamma_2, \Delta)) - \Upsilon(\sigma, S, \Delta_1) + q).
\end{aligned}$$

The constant is nonnegative because of the corollary on the $\cdot = \cdot \succ \cdot \,|\, \cdot$ relation, Equation 5.5.

The induction hypothesis then yields m_0 such that $m, S, \sigma \vdash^{g,t} e_1 \rightsquigarrow v_o, \sigma_0, m_0$ with

$$\begin{aligned}
m_0 \geq n_0 + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, (\Gamma_1, \Delta_1), l) - \Upsilon'(\sigma_0, v_0, T_0, l)\right\} \\
+ \Upsilon(\sigma_0, v_0, T_0) + (\Upsilon(\sigma, S, (\Gamma_2, \Delta)) - \Upsilon(\sigma, S, \Delta_1) + q)
\end{aligned}$$

To form the precondition for applying the induction hypothesis to e_2 we note that the separation condition (Definition 2.3) guarantees that

$$\mathcal{R}(\sigma, S \restriction \text{dom}(\Delta_1)) \cap \mathcal{R}(\sigma', v_0) = \emptyset,$$

and so $\Upsilon'(\sigma, S, \Delta_1, l) = 0$ for all $l \in \mathcal{R}(\sigma', v_0)$, and $\Upsilon'(\sigma', v_0, T_0, l) = 0$ for all $l \in \mathcal{R}(\sigma, S \upharpoonright \mathrm{dom}(\Delta_1))$ by Lemma 5.7. Thus we can separate out the give-back potential for Δ, and then apply Equation 5.7 to form the precondition:

$$
\begin{aligned}
m_0 \geq\ & n_0 + \sum_{l \in \mathcal{R}(\sigma, S \upharpoonright \mathrm{dom}(\Delta_1))} \left(\Upsilon'(\sigma, S, \Gamma_1, l) + \Upsilon'(\sigma, S, \Delta_1, l)\right) \\
& + \sum_{l \notin \mathcal{R}(\sigma, S \upharpoonright \mathrm{dom}(\Delta_1))} \max\left\{0, \Upsilon'(\sigma, S, \Gamma_1, l) - \Upsilon'(\sigma_0, v_0, T_0, l)\right\} \\
& + \Upsilon(\sigma_0, v_0, T_0) + \Upsilon(\sigma, S, (\Gamma_2, \Delta)) - \Upsilon(\sigma, S, \Delta_1) + q \\
\geq\ & n_0 + \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma_0, S, \Delta_1, l) + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma_1, l) - \Upsilon'(\sigma, v_0, T_0, l)\right\} \\
& + \Upsilon(\sigma_0, v_0, T_0) + \Upsilon(\sigma, S, (\Gamma_2, \Delta)) - \Upsilon(\sigma, S, \Delta_1) + q \\
\geq\ & n_0 + \Upsilon(\sigma, S, (\Gamma_2, \Delta_2)) + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma_1, l) - \Upsilon'(\sigma_0, v_0, T_0, l)\right\} \\
& + \Upsilon(\sigma_0, v_0, T_0) + q \\
\geq\ & n_0 + \Upsilon(\sigma_0, S[x \mapsto v_0], ((\Gamma_2, \Delta_2) \upharpoonright \mathrm{FV}(e_2), x : T_0)) \\
& + \left(\sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma_1, l) - \Upsilon'(\sigma_0, v_0, T_0, l)\right\} + \Upsilon(\sigma, S, (\Gamma_2, \Delta_2) \setminus \mathrm{FV}(e_2)) + q\right)
\end{aligned}
$$

using the benign sharing condition

$$\sigma \upharpoonright \mathcal{R}(\sigma, S_{e_2}) = \sigma_0 \upharpoonright \mathcal{R}(\sigma, S_{e_2}), \tag{2.2}$$

where $S_{e_2} = S \upharpoonright (\mathrm{FV}(e_2) \setminus x)$, and Lemma 5.7 to account for changes in the state in the last step. Note that we need to separate out the unused variables, because the benign sharing conditions only make guarantees for $\mathrm{FV}(e_2)$. We can apply Lemma 5.8 to remove the unused variables from the typing of e_2.

Finally, the induction hypothesis on e_2 yields an m' such that

$$
\begin{aligned}
m' \geq\ & n' + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma_0, S[x \mapsto v_0], ((\Gamma_2, \Delta_2) \upharpoonright \mathrm{FV}(e_2), x : T_0), l) - \Upsilon'(\sigma', v, T, l)\right\} \\
& + \Upsilon(\sigma', v, T) \\
& + \left(\sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma_1, l) - \Upsilon'(\sigma_0, v_0, T_0, l)\right\} + \Upsilon(\sigma, S, (\Gamma_2, \Delta_2) \setminus \mathrm{FV}(e_2)) + q\right) \\
\geq\ & n' + \sum_{l \in \mathsf{loc}} \max\left\{0, \begin{array}{ll} \Upsilon'(\sigma, S, (\Gamma_2, \Delta_2), l) & + \Upsilon'(\sigma_0, v_0, T_0, l) - \Upsilon'(\sigma', v, T, l) \\ & + \Upsilon'(\sigma, S, \Gamma_1, l) - \Upsilon'(\sigma_0, v_0, T_0, l) \end{array}\right\} \\
& + \Upsilon(\sigma', v, T) + q \\
\geq\ & n' + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, (\Gamma_1, \Gamma_2, \Delta), l) - \Upsilon'(\sigma', v, T, l)\right\} + \Upsilon(\sigma', v, T) + q
\end{aligned}
$$

as required, using benign sharing and Lemma 5.7 to change back from σ_0 to σ, and Equation 5.6 to change Δ_2 to Δ.

IF. We only remove the boolean (which has zero potential by definition) from the context, so m must be large enough to invoke the induction hypothesis on the appropriate branch. As the give-back potential $\Upsilon'(\sigma, c, \mathsf{bool}, l) = 0$ as well, the m' from the induction hypothesis is sufficient.

PAIRELIM. The value of the pair is $S(x) = (v_1, v_2)$ for some values v_1, v_2 and the type is $T = T_1 \otimes T_2$ for some T_1, T_2. The definition of Υ is linear for pairs, so

$$\begin{aligned} m &\geq n + \Upsilon(\sigma, S, \Gamma \setminus x) + \Upsilon(\sigma, (v_1, v_2), T_1 \otimes T_2) + q \\ &\geq n + \Upsilon(\sigma, S[x_1 \mapsto v_1, x_2 \mapsto v_2], (\Gamma \setminus x, x_1 : T_1, x_2 : T_2)) + q \end{aligned}$$

and we can apply the induction hypothesis. The resulting m' will be sufficiently large because Υ' is also linear with respect to pairs.

SUMELIM with E-MATCHINL. The typing rule requires the constraint $n_l = n + k_l$ to be satisfied. Hence we establish that m is sufficient to apply the induction hypothesis on e_l using the definition of Υ on sum types:

$$\begin{aligned} m &\geq n + \Upsilon(\sigma, S, \Gamma \setminus x) + \Upsilon(\sigma, \mathsf{inl}(v'), (T_l, k_l) + (T_r, k_r)) + q \\ &= n + \Upsilon(\sigma, S, \Gamma \setminus x) + \Upsilon(\sigma, v', T_l) + k_l + q \\ &= n_l + \Upsilon(\sigma, S[x_l \mapsto v'], (\Gamma \setminus x, x_l : T_l)) + q. \end{aligned}$$

By the induction hypothesis there is an m' such that

$$\begin{aligned} m' &\geq n' + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S[x_l \mapsto v'], (\Gamma \setminus x, x_l : T_l), l) - \Upsilon'(\sigma', v, T, l)\right\} + \Upsilon(\sigma', v, T) + q \\ &= n' + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma', v, T, l)\right\} + \Upsilon(\sigma', v, T) + q \end{aligned}$$

as required, because $\Upsilon'(\sigma, \mathsf{inl}(v'), (T_l, k_l) + (T_r, k_r), l) = \Upsilon'(\sigma, v', T_l, l)$ by the definition of Υ'.

SUMELIM with E-MATCHINR. Analogous to E-MATCHINL.

MATCH and CASE-GB with E-MATCHN. We use the constraint $n_i = n + k_i$ (the size of a constructor represented by null is zero by definition) to show that

$$\begin{aligned} m &\geq n + \Upsilon(\sigma, S, \Gamma \setminus x) + \Upsilon(\sigma, \mathsf{null}, ty(\overline{k \rightsquigarrow k'})) + q \\ &\geq n + k_i + \Upsilon(\sigma, S, \Gamma \setminus x) + q \\ &= n_i + \Upsilon(\sigma, S, \Gamma \setminus x) + q. \end{aligned}$$

Then the induction hypothesis can be applied to e_i. The resulting m' is sufficiently large because $\Upsilon'(\sigma, \mathsf{null}, ty(\overline{k \rightsquigarrow k'}), l) = 0$ by definition, so $\Upsilon'(\sigma, S, \Gamma \setminus x, l) = \Upsilon'(\sigma, S, \Gamma, l)$.

MATCH and CASE-GB with E-MATCH. The value of x is $S(x) = l$ for some $l \in \mathsf{loc}$, and $\sigma(l) = (c_i, v_1, \ldots, v_p)$.

The deallocation will provide an extra $\mathrm{size}(c_i)$ units of memory to execute e_i with. The constraint $n_i = n + k_i + \mathrm{size}(c_i)$ in the typing rule reflects this. Thus we can show that m plus the deallocated memory is large enough to apply the induction hypothesis:

$$
\begin{aligned}
& m + \mathrm{size}(c_i) \\
& \quad \geq n + \Upsilon(\sigma, S, \Gamma) + q + \mathrm{size}(c_i) \\
& \quad \geq n + \Upsilon(\sigma, S, \Gamma \setminus x) + \Upsilon(\sigma, S(x), ty(\overline{k \rightsquigarrow k'})) + q + \mathrm{size}(c_i) \\
& \quad \geq n + k_i + \mathrm{size}(c_i) + \Upsilon(\sigma, S, \Gamma \setminus x) + \sum_{j=1}^{p} \Upsilon(\sigma_i, v_j, T_j) + q \\
& \quad \geq n_i + \Upsilon(\sigma, S, \Gamma \setminus x) + \Upsilon(\sigma_i, [x_1 \mapsto v_1, \ldots, x_p \mapsto v_p], (x_1 : T_1, \ldots, x_p : T_p)) + q \\
& \quad = n_i + \Upsilon(\sigma_i, S_i, \Gamma_i) + (\Upsilon(\sigma, S, \Gamma \setminus \mathrm{FV}(e_i)) + q)
\end{aligned}
$$

where $\sigma_i = \sigma \setminus l$, $S_i = (S \setminus x)[x_1 \mapsto v_1, \ldots, x_p \mapsto v_p]$ and

$$\Gamma_i = ((\Gamma \setminus x) \restriction \mathrm{FV}(e_i)), x_1 : T_1, \ldots, x_p : T_p.$$

The last step uses the benign sharing condition

$$l' \notin \mathcal{R}(\sigma, S[x_1 \mapsto v_1, \ldots, x_p \mapsto v_p] \restriction \mathrm{FV}(e_i)) \tag{2.1}$$

together with Lemma 5.7 to remove the dead location l' from the calculation of the potential. As with LET-GB, we need to separate out the potential of any unused variables because the condition is only valid for $\mathrm{FV}(e_i)$. Again, we can remove such variables from the typing of e_i by Lemma 5.8.

For the bound after execution, we use the second constraint, $n'_i = n' + k'_i$,

$$
\begin{aligned}
m' & \geq n'_i + \sum_{l' \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma_i, S_i, \Gamma_i, l') - \Upsilon'(\sigma', v, T, l')\right\} + \Upsilon(\sigma', v, T) + (\Upsilon(\sigma, S, \Gamma \setminus \mathrm{FV}(e_i)) + q) \\
& \geq n' + k'_i + \sum_{l' \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma \setminus x, l') + \sum_{i=1}^{p} \Upsilon'(\sigma, v_j, T_j, l') - \Upsilon'(\sigma', v, T, l')\right\} \\
& \qquad + \Upsilon(\sigma', v, T) + q \\
& \geq n' + \sum_{l' \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma, l') - \Upsilon'(\sigma', v, T, l')\right\} + \Upsilon(\sigma', v, T) + q
\end{aligned}
$$

using the definition of $\Upsilon'(\sigma, l', ty(\overline{k \rightsquigarrow k'}), l')$ to form the give-back potential for x.

MATCH and CASE′-GB with E-MATCH′ or E-MATCHN′. Similar to the previous two cases, except that there is no deallocation. Thus the size(c_i) terms disappear from both the constraint and the operational semantics, and l is not removed from σ. In all other respects, the argument is the same.

FUN-TAIL. As the whole program is well-typed and the FUN-TAIL rule requires that there is a signature

$$\Sigma(f) = T'_1, \ldots, T'_p, k \to T', k' \,|\, \Phi'$$

for the function called, there must be a typing derivation ending in the judgement

$$x_1 : T'_1, \ldots, x_p : T'_p, k \vdash^{f,\mathsf{true}}_{\Sigma,F'} e_f : T', k' \,|\, \Phi'$$

where e_f is the body of f (up to the replacement of variable names). Using the substitution ρ, there is also a derivation ending in

$$x_1 : T_1, \ldots, x_p : T_p, \rho(k) \vdash^{f,\mathsf{true}}_{\Sigma,F} e_f : T, \rho(k') \,|\, \rho(\Phi')$$

which we will use for the induction.

Now we can show that m is large enough to execute e_f, after the allocation of a stack frame:

$$\begin{aligned} m &\geq n + \Upsilon(\sigma, S, \Gamma) + q \\ &\geq \rho(k) + \mathrm{stack}'(g, f, t) + \Upsilon(\sigma, S, (x_1 : T_1, \ldots, x_p : T_p)) \\ &\quad + \big(n - \rho(k) - \mathrm{stack}'(g, f, t) + \Upsilon(\sigma, S, \Gamma \setminus \{x_1, \ldots, x_p\}) + q\big) \end{aligned}$$

Note that we incorporate the unused portion of n and the potential from the unused variables in the context into the constant for the induction to guarantee that they are available in m'. The constraint on n in the typing rule guarantees that the constant will be non-negative. The induction hypothesis plus the second

constraint $n - \rho(k) + \rho(k') \geq n'$ thus gives

$$\begin{aligned} m' &\geq \rho(k') + \text{stack}'(g,f,t) \\ &\quad + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, (x_1 : T_1, \ldots, x_p : T_p), l) - \Upsilon'(\sigma', v, T, l)\right\} + \Upsilon(\sigma', v, T) \\ &\quad + \left(n - \rho(k) - \text{stack}'(g,f,t) + \Upsilon(\sigma, S, \Gamma \setminus \{x_1, \ldots, x_p\}) + q\right) \\ &\geq n - \rho(k) + \rho(k') + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma', v, T, l)\right\} + \Upsilon(\sigma', v, T) + q \\ &\geq n' + \sum_{l \in \mathsf{loc}} \max\left\{0, \Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma', v, T, l)\right\} + \Upsilon(\sigma', v, T) + q \end{aligned}$$

as required.

FUNDEF-TAIL. Similar to FUN-TAIL, except that we do not require the substitution ρ. □

We can also show that the overall bound on a program is sound:

Corollary 5.10. *Suppose a well typed program has an initial function f, and arguments for f are given as values $v_1, \ldots, v_p$ with an initial store σ. If*

$$\Sigma(f) = T_1, \ldots, T_p, k \to T', k' \mid \Phi$$

then any execution of $f(v_1, \ldots, v_p)$ will require at most

$$\Upsilon(\sigma, [x_1 \mapsto v_1, \ldots, x_p \mapsto v_p], (x_1 : T_1, \ldots, x_p : T_p)) + \text{stack}(f) + k$$

units of memory, for any assignment of nonnegative rationals to constraint variables which satisfies Φ.

Proof. The definition of Υ implies that the values v_i and store σ are consistent with the types in $\Sigma(f)$. Thus by FUN-TAIL we have

$$x_1 : T_1, \ldots, x_p : T_p, n \vdash^{\mathsf{initial},\mathsf{false}}_{\Sigma,F} f(x_1, \ldots, x_p) : T', n' \mid \Phi \cup \{n \geq k + \text{stack}(f), n - k + k' \geq n'\}$$

and so Theorem 5.9 guarantees that any execution with at least

$$\Upsilon(\sigma, [x_1 \mapsto v_1, \ldots, x_p \mapsto v_p], (x_1 : T_1, \ldots, x_p : T_p)) + \text{stack}(f) + k$$

units of free memory will not run out of memory. □

5.2.3 Soundness of previous analyses

We can establish the soundness of the analysis in Chapter 4 by noting that this system is a strict extension of it.

Lemma 5.11. *Any typing in the system presented in Chapter 4 corresponds to a typing in the 'give-back' system above. Therefore the soundness results for the bounds on expressions (Theorem 5.9) and programs (Corollary 5.10) hold.*

Proof. We add give-back annotations to the types in the derivation but fix them to be zero. When typing let expressions we use LET-GB, taking Δ to be the empty context (so that all contraction is handled by the SHARE rule). We also replace CASE and CASE′ by CASE-GB and CASE′-GB respectively. The resulting constraint sets are then equivalent to those in the original derivation.

We can then apply Theorem 5.9 and Corollary 5.10. □

This result also extends to the Hofmann-Jost system presented in Chapter 2 by taking $\mathrm{stack}(f) = 0$ for all functions f, proving Theorem 2.4 and Corollary 2.5.

5.2.4 Partial executions

The soundness result can be extended to programs which do not terminate or which fail at a match expression using the augmented operational semantics from Section 2.1.3. The extended theorem will show that all such partial executions also respect the inferred bound.

Theorem 5.12. *If an expression e in a well-typed program has a typing*

$$\Gamma, n \vdash^{g,t}_{\Sigma,F} e : T, n' \,|\, \Phi$$

with an assignment of nonnegative rationals to constraint variables which satisfies the constraints in g's signature, and an evaluation $S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma'$ in the augmented semantics which satisfies the benign sharing conditions, and $\Upsilon(\sigma, S, \Gamma)$ is defined, then for any $q \in \mathbb{Q}^+$ and $m \in \mathbb{N}$ such that

$$m \geq n + \Upsilon(\sigma, S, \Gamma) + q$$

we have $m, S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma', m'$ where either $v = \mathsf{halted}$, or

$$m' \geq n' + \sum_{l \in \mathsf{loc}} \max\{0, \Upsilon'(\sigma, S, \Gamma, l) - \Upsilon'(\sigma', v, T, l)\} + \Upsilon(\sigma', v, T) + q.$$

Proof. The proof uses the same simultaneous induction as for Theorem 5.9, except that we do not need to check m' when $v = \mathsf{halted}$. We only need to consider the three new rules in the extended operational semantics (Figure 2.4):

E-STOP. Immediate because $v = \mathsf{halted}$.

LET-GB with E-LET. The proof in Theorem 5.9 suffices because we have only added the extra premise that $v_0 \neq \mathsf{halted}$.

LET-GB with E-STOPLET. We can use the proof for LET-GB from Theorem 5.9 to show that we can invoke the induction hypothesis on e_1. The result then follows because $v = \mathsf{halted}$. □

We can also exploit the above soundness theorem to show that the analysis can be used as a rather limited termination analysis. Consider the results of a stack-only analysis *without* tail-call optimisation. A non-terminating program must feature arbitrarily large partial executions, and the only way to form such executions is by arbitrarily large recursive call chains. However, any bound obtained from the analysis bounds the call depth of all partial evaluations by the soundness theorem, so arbitrarily large partial executions are impossible. Hence if the analysis yields a bound the program must terminate. However, this termination analysis is rather weak because it requires that the call depth has a linear bound in terms of the size of the program's input.

5.2.5 Inference and implementation

The inference process has the same form as before (in Section 2.2.5); most of the rules are syntax directed and we use linear programming techniques to minimise the potential, which provides the bound. As before, the non-syntax directed rules are function application and contraction, and the choice between FUN-TAIL and FUNDEF-TAIL is decided by membership of the set of defined functions, F.

The key difference is that we use LET-GB for contraction wherever possible. We can decide if LET-GB can be used by checking the separation information from the safety analysis. The SHARE rule is used for other instances of contraction (such as `andlists(l,l)`) and weakening as before. If LET-GB is used for contraction and SHARE used for weakening of the context for the second subexpression, then the resulting constraints allow strictly more solutions than SHARE alone. Using this technique wherever LET-GB can be used for contraction can only improve the bound.

In the implementation, the type inference combines the LET-GB and SHARE rules, but we have presented them separately to simplify the soundness proof above.

We take the linear program from the generated constraints and the constraints from the two conventions on give-back annotations ($k \geq k'$ or $k' = 0$ depending upon the circumstances), and solve them using linear programming techniques as before.

The number of constraints generated by inference in our new analysis is within a constant factor of the Hofmann-Jost analysis. The extra constraints either accompany an existing one (that is, constraints on give-back annotations which correspond to constraints on the 'normal' annotations), or are the result of one of our conventions on annotations. At most one constraint per annotation is required, so the increase in constraints is linear.

The inference process has been implemented in an extension of Jost's implementation (Jost, 2004b). It includes the other systems from the preceding chapters and an option to perform the CPS transformation for comparison. The separation information which is used to decide how to perform contraction is supplied as part of the program text. Konečný's implementation of his DEEL system (Konečný, 2003) has been adapted to provide this information. Details on applying the analysis to the heap sort example can be found in Appendix A.

5.2.6 Examples

As an extended example of the give-back analysis we present a full typing for the `andlists2` program from Example 5.2. Figure 5.6 shows a typing of the prerequisite `andlists` function. The difference between this function signature and one from the previous analysis is that we can express the potential after evaluation in terms of the argument that supplied the potential required for evaluation. In particular, the three solutions desired for the stack space of `andlists` from Section 5.2 can be obtained as solutions;

$$\mathsf{boollist}(1 \rightsquigarrow 1), \mathsf{boollist}(0 \rightsquigarrow 0), 0 \to \mathsf{boollist}(0 \rightsquigarrow 0), 0,$$
$$\mathsf{boollist}(1 \rightsquigarrow 0), \mathsf{boollist}(0 \rightsquigarrow 0), 0 \to \mathsf{boollist}(1 \rightsquigarrow 0), 0,$$
$$\mathsf{boollist}(1 \rightsquigarrow \tfrac{1}{2}), \mathsf{boollist}(0 \rightsquigarrow 0), 0 \to \mathsf{boollist}(\tfrac{1}{2} \rightsquigarrow 0), 0,$$

where the free memory afterwards is described respectively in terms of the first argument, the result, and half of each. Only the middle form could be obtained in the previous analysis. Note that the use of the CASE′-GB rule for `l1` produces two sym-

metric constraints (Φ_7): one extracts the potential attached to the head of the list which will be used for the recursive call; the other restores the corresponding give-back potential, and any remainder provides potential for the result. There is no contraction in `andlists`, so Δ is always empty in the uses of the LET-GB rule.

Figure 5.7 presents the typing for `andlists2`, using the `andlists` typing for the two function calls. The LET-GB rule allows us to use the give-back potential of the first instance of `l1` as the potential for its second use. Hence we can use the first type signature for `andlists` above for the first function call and any of the three for the second, reusing the potential assigned to `l1`. This yields the three solutions for `andlists2` from Section 5.2:

$$\mathsf{boollist}(1 \rightsquigarrow 1), \mathsf{boollist}(0 \rightsquigarrow 0), \mathsf{boollist}(0 \rightsquigarrow 0), 1 \to \mathsf{boollist}(0 \rightsquigarrow 0) \times \mathsf{boollist}(0 \rightsquigarrow 0), 1$$
$$\mathsf{boollist}(1 \rightsquigarrow 0), \mathsf{boollist}(0 \rightsquigarrow 0), \mathsf{boollist}(0 \rightsquigarrow 0), 1 \to \mathsf{boollist}(0 \rightsquigarrow 0) \times \mathsf{boollist}(1 \rightsquigarrow 0), 1$$
$$\mathsf{boollist}(1 \rightsquigarrow \tfrac{1}{2}), \mathsf{boollist}(0 \rightsquigarrow 0), \mathsf{boollist}(0 \rightsquigarrow 0), 1 \to \mathsf{boollist}(0 \rightsquigarrow 0) \times \mathsf{boollist}(\tfrac{1}{2} \rightsquigarrow 0), 1$$

The side condition on the LET-GB rule requires that `l1` is separate from the result of `andlists(l1,l2)`. It is satisfied because the result is freshly allocated.

We can also bound the total memory usage using the give-back analysis. Note that the constraints in $\Psi_{\succ}$ allow some of the potential to be split between the uses of `l1` (like SHARE) as well as using the given-back potential. Thus the heap memory requirements are simply added to the bound.

We claimed in Chapter 4 that our extended analysis would provide some assistance for the problem with accumulating parameters discussed there.

Example 5.13. Recall the `revapp` reverse-and-append function from Example 4.6. Previously, we could not show that the stack space required without tail-call optimisation was available afterwards. We could only obtain function signatures such as

$$\texttt{revapp} : \mathsf{boollist}(k), \mathsf{boollist}(0), 0 \to \mathsf{boollist}(0), 0,$$

where $k = \mathrm{stack}(\texttt{revapp})$. With the give-back analysis we can get function signatures of the form

$$\texttt{revapp} : \mathsf{boollist}(k \rightsquigarrow k), \mathsf{boollist}(0 \rightsquigarrow 0), 0 \to \mathsf{boollist}(0 \rightsquigarrow 0), 0,$$

showing that the stack space is free afterwards through the give-back potential of the first argument. (We still cannot solve the original problem, namely that we cannot assign the corresponding potential to the result.)

$$\mathcal{D}_{\text{and}} = \dfrac{\dfrac{}{\texttt{h2} : \mathsf{bool}, n_2 \vdash^{\mathsf{f}}_{\Sigma,\emptyset} \texttt{h2} : \mathsf{bool}, n_3 \mid \Phi_3}\ \textsc{Var} \qquad \dfrac{}{\texttt{h2} : \mathsf{bool}, n_2 \vdash^{\mathsf{f}}_{\Sigma,\emptyset} \texttt{false} : \mathsf{bool}, n_3 \mid \Phi_3}\ \textsc{Bool}}{\texttt{h1} : \mathsf{bool}, \texttt{h2} : \mathsf{bool}, n_2 \vdash^{\mathsf{f}}_{\Sigma,\emptyset} \texttt{if} \cdots : \mathsf{bool}, n_3 \mid \Phi_3}\ \textsc{If}$$

$$\mathcal{D} = \dfrac{\mathcal{D}_{\text{and}} \qquad \dfrac{\dfrac{\dfrac{}{\texttt{t1} : T_1, \texttt{t2} : T_2, n_3 \vdash^{\mathsf{f}}_{\Sigma,\emptyset} \texttt{andlists(t1,t2)} : T_3, n_4 \mid \Phi_4}\ \textsc{FunDef-Tail} \qquad \dfrac{}{\texttt{h} : \mathsf{bool}, \texttt{t} : T_3, n_4 \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{cons(h,t)} : T_3, n_2' \mid \Phi_5}\ \textsc{Construct}}{\texttt{h} : \mathsf{bool}, \texttt{t1} : T_1, \texttt{t2} : T_2, n_3 \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{let t} \cdots : T_3, n_2' \mid \Phi_{4,5}}\ \textsc{Let-GB}}{\texttt{h1} : \mathsf{bool}, \texttt{h2} : \mathsf{bool}, \texttt{t1} : T_1, \texttt{t2} : T_2, n_2 \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{let h} \cdots : T_3, n_2' \mid \Phi_{3,4,5}}\ \textsc{Let-GB}}{\texttt{h1} : \mathsf{bool}, \texttt{t1} : T_1, n_1 \vdash T_2 \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{cons(h2,t2)}' \to \cdots : T_3, n_1' \mid \Phi_{3,\ldots,6}}\ \textsc{Case}'\textsc{-GB}$$

$$\dfrac{\dfrac{\dfrac{}{\texttt{l2} : T_2, n \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{nil} : T_3, n' \mid \Phi_1}\ \textsc{Construct}}{\texttt{l2} : T_2, n \vdash T_1 \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{nil} \to \texttt{nil} : T_3, n' \mid \Phi_1}\ \textsc{Case}'\textsc{-GB} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{h1} : \mathsf{bool}, \texttt{t1} : T_1, n_1 \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{nil} : T_3, n_1' \mid \Phi_2}\ \textsc{Construct}}{\texttt{h1} : \mathsf{bool}, \texttt{t1} : T_1, n_1 \vdash T_2 \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{nil} : T_3, n_1' \mid \Phi_2}\ \textsc{Case}'\textsc{-GB} \qquad \mathcal{D}}{\texttt{h1} : \mathsf{bool}, \texttt{t1} : T_1, \texttt{l2} : T_2, n_1 \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{match l2} \cdots : T_3, n_1' \mid \Phi_{2,\ldots,6}}\ \textsc{Match}}{\texttt{l2} : T_2, n \vdash T_1 \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{cons(h1,t1)}' \to \cdots : T_3, n' \mid \Phi_{2,\ldots,7}}\ \textsc{Case}'\textsc{-GB}}{\texttt{l1} : T_1, \texttt{l2} : T_2, n \vdash^{\mathsf{t}}_{\Sigma,\emptyset} \texttt{match l1} \cdots : T_3, n' \mid \Phi_{1,\ldots,7}}\ \textsc{Match}$$

$$T_i = \mathsf{boollist}(k_i \rightsquigarrow k_i') \qquad \Phi_{i_1,\ldots,i_m} = \Phi_{i_1} \cup \cdots \cup \Phi_{i_m}$$

$$\Phi_1 = \{n \geq n'\},\ \Phi_2 = \{n_1 \geq n_1'\},\ \Phi_3 = \{n_2 \geq n_3\},$$

$$\Phi_4 = \{n_3 \geq n + \mathrm{stack}'(\texttt{andlists}, \texttt{andlists}, \mathsf{false}), n_3 - n + n' \geq n_4\},$$

$$\Phi_5 = \{n_4 \geq \mathrm{size}(\mathsf{cons}) + k_3 + n_2'\},$$

$$\Phi_6 = \{n_2 = n_1 + k_2, n_2' = k_2' + n_1'\},\ \Phi_7 = \{n_1 = n + k_1, n_1' = k_1' + n'\}.$$

$$\Sigma = \begin{bmatrix} \texttt{nil} \mapsto \forall k, k'.\mathsf{boollist}(k \rightsquigarrow k') \\ \mathsf{cons} \mapsto \forall k, k'.\mathsf{bool}, \mathsf{boollist}(k \rightsquigarrow k'), k \rightsquigarrow k' \to \mathsf{boollist}(k \rightsquigarrow k') \\ \texttt{andlists} \mapsto T_1, T_2, n \to T_3, n' \end{bmatrix}$$

Figure 5.6: `andlists` typing in the give-back analysis

$$\mathcal{D}_1 = \frac{}{\texttt{l1} : T_{11}, \texttt{l2} : T_2, n_0 \vdash^{\mathsf{f}}_{\Sigma',A} \texttt{andlists}(\texttt{l1},\texttt{l2}) : T_4, n_1 \,|\, \Psi_1}\ \textsc{Fun-Tail}$$

$$\mathcal{D}_2 = \frac{}{\texttt{l1} : T_{12}, \texttt{l3} : T_3, n_1 \vdash^{\mathsf{f}}_{\Sigma',A} \texttt{andlists}(\texttt{l1},\texttt{l3}) : T_5, n_2 \,|\, \Psi_2}\ \textsc{Fun-Tail}$$

$$\mathcal{D}_3 = \frac{}{\texttt{r1} : T_4, \texttt{r2} : T_5, n_2 \vdash^{\mathsf{t}}_{\Sigma',A} (\texttt{r1},\texttt{r2}) : T_4 \otimes T_5, n' \,|\, \{n_2 \geq n'\}}\ \textsc{Pair}$$

$$\mathcal{D}_4 = \frac{\mathcal{D}_2 \quad \mathcal{D}_3}{\texttt{l1} : T_{12}, \texttt{l3} : T_3, \texttt{r1} : T_4, n_1 \vdash^{\mathsf{t}}_{\Sigma',A} \texttt{let r2} \cdots : T_4 \otimes T_5, n' \,|\, \Psi_2 \cup \{n_2 \geq n'\}}\ \textsc{Let-GB}$$

$$\frac{\texttt{l1} : T_1 = \texttt{l1} : T_{11} \succ \texttt{l1} : T_{12} \,|\, \Psi_\succ \quad \mathcal{D}_1 \quad \mathcal{D}_4}{\texttt{l1} : T_1, \texttt{l2} : T_2, \texttt{l3} : T_3, n_0 \vdash^{\mathsf{t}}_{\Sigma',A} \texttt{let r1} \cdots : T_4 \otimes T_5, n' \,|\, \Psi_1 \cup \Psi_2 \cup \{n_2 \geq n'\} \cup \Psi_\succ}\ \textsc{Let-GB}$$

$$T_i = \mathsf{boollist}(k_i \rightsquigarrow k_i') \qquad A = \{\texttt{andlists}\}$$

$$\Psi_1 = \rho_1(\Phi_{1,\ldots,7}) \cup \{n_0 \geq \rho_1(n) + \mathrm{stack}'(\texttt{andlists2}, \texttt{andlists}, \mathsf{false}),\ n_0 - \rho_1(n) + \rho_1(n') \geq n_1\},$$

$$\Psi_2 = \rho_2(\Phi_{1,\ldots,7}) \cup \{n_1 \geq \rho_2(n) + \mathrm{stack}'(\texttt{andlists2}, \texttt{andlists}, \mathsf{false}),\ n_1 - \rho_2(n) + \rho_2(n') \geq n_2\},$$

$$\Psi_\succ = \{k_1 \geq \rho_1(k_1),\ k_1 - \rho_1(k_1) + \rho_1(k_1') \geq \rho_2(k_1),\ \rho_2(k_1') \geq k_1'\}.$$

$$\Sigma' = \Sigma \cup [\texttt{andlists2} \mapsto T_1, T_2, T_3, n_0 \to T_4 \otimes T_5, n_0']$$

Figure 5.7: `andlists2` typing in the give-back analysis

Sadly, for this particular example that knowledge is of little use — the original list is destroyed by `revapp`, and so the given-back potential cannot be reused. However, non-destructive accumulating functions can benefit from the give-back extension in general. For instance, if we had a non-destructive version of `revapp` and some later bound expressed in terms of the first argument for `revapp`, then the overall bound would take into account the reuse of the stack memory.

Chapter 6

Bounding in terms of depth

We now return to the two problems discussed in Chapter 4 that we promised to investigate: providing stack space bounds in terms of the depth of data structures, and extending the form the bounds can take to include maxima. Our motivation is to provide tighter bounds on stack space usage, and in particular, for tree-structured data, and for larger programs where a sequence of function calls requires stack space which is the maximum of their individual bounds. However, the new analysis will still have no knowledge of 'global' invariants, such as balanced trees.

We intend to retain many of the features of the Hofmann-Jost analysis in our new system. We assign potential to data structures via their types; use the type system to produce constraints on the potential that express 'local' changes in the potential, corresponding to allocation or transferring potential from one data structure to another (to obtain bounds in terms of the input size rather than an intermediate variable's size); and we will continue to use linear programming to solve constraints provided by the type system.

In this chapter we will present the type system for the analysis, prove its soundness with respect to the operational semantics and discuss constraint solving and some examples. The type system has a more substructural flavour than those in previous chapters, and so we will require some additional inference to decide where and how to use the non-syntax-directed rules. We will present an inference algorithm in the following chapter.

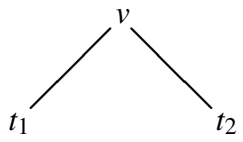
Note that we only consider stack space in this analysis. This is partly because bounds expressed in terms of depth are of little use when assessing heap space requirements, but also because it considerably simplifies the typing of let expressions, as we will discuss below. Some preliminary ideas on using a similar system for heap space

analysis are set out in Section 9.2.3.

6.1 Informal description of the type system

The potential functions for our new analysis must differ from the definitions in the previous systems by calculating amounts proportional to the depth for algebraic datatypes, and also taking the maximum potential of several subcontexts where appropriate rather than always summing them.

Consider the topmost node of a binary tree with boolean values, t, which has subtrees t_1 and t_2:



The depth of the whole tree is $|t|_{\mathrm{d}} = \max\{|t_1|_{\mathrm{d}}, |t_2|_{\mathrm{d}}\} + 1$. Thus it is natural to study depth and maxima together. Moreover, when the tree t is unfolded by a match expression the (unannotated) typing context for the subexpression will contain

$$t_1 : \mathsf{booltree},\ t_2 : \mathsf{booltree},\ v : \mathsf{bool},$$

and the potential of the whole context must involve the maximum of the potential of t_1 and t_2, plus a fixed amount of potential for the top level of t.

Our approach to defining the new type system and potential functions is inspired by O'Hearn's Bunched Typing (O'Hearn, 2003). Bunched typing introduces tree-structured typing contexts by using two context formers, ',' and ';'. The use of various structural rules, such as contraction, can be restricted depending upon the context former involved, and some rearrangement of the tree structure may be allowed. The main application presented for bunched typing is to denote resource sharing (especially for heap cells). There ',' can be interpreted as 'the subcontexts share no resources' and ';' as 'the subcontexts may share resources'. Thus contraction is allowed for ';' but not ',' because the two resulting subcontexts share all resources. It is also easy to see that the distributivity rule for contexts,

$$\Gamma, (\Delta; \Delta') \cong (\Gamma, \Delta); (\Gamma, \Delta'),$$

is sound in this application.

For our purposes, the ',' context former will mean 'sum the potential of the subcontexts' and ';' will mean 'take the maximum of the subcontexts' potential.' Thus the context for the unfolded tree node above becomes

$$(t_1 : \mathsf{booltree}(k);\ t_2 : \mathsf{booltree}(k);\ v : \mathsf{bool}),\ k$$

where k is the fixed amount of potential associated with the top node of the tree, and so the total potential is $\max\{k \times |t_1|_{\mathrm{d}}, k \times |t_2|_{\mathrm{d}}, 0\} + k$, as desired (where the 0 is the potential of $v : \mathsf{bool}$). Note that we will allow fixed amounts of potential to appear anywhere in the context, even though no variables are involved. This allows us to express bounds such as $\max\{|x|_{\mathrm{d}} + k, |y|_{\mathrm{d}}\}$ (using the context $(x : T_1, k); y : T_2$).

To exploit this correspondence between the depth of datatypes and using context structure to take sums and maxima of potential, the potential functions for algebraic datatypes are defined by putting contexts in the constructor's signature. For example, the signature for a node in the previous systems would be

$$\Sigma_{\mathrm{prev}}(\mathsf{node}) = \forall k.\mathsf{booltree}(k),\ \mathsf{booltree}(k),\ \mathsf{bool},\ k \to \mathsf{booltree}(k),$$

but here we use

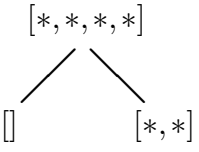
$$\Sigma_{\mathrm{depth}}(\mathsf{node}) = \forall k.(t_1 : \mathsf{booltree}(k);\ t_2 : \mathsf{booltree}(k);\ v : \mathsf{bool}),\ k \to \mathsf{booltree}(k),$$

reflecting the potential function required[1].

For nested types the potential is not simply the depth, but is influenced by the potential of the 'contents'. For example, consider a type of binary trees with unit lists at the nodes:

$$\Sigma = \left[\begin{array}{l} \mathtt{nil} \mapsto \forall k_l.0 \to \mathsf{list}(k_l) \\ \mathtt{cons} \mapsto \forall k_l.(h : 1; t : \mathsf{list}(k_l)), k_l \to \mathsf{list}(k_l) \\ \mathtt{leaf} \mapsto \forall k_t, k_l.0 \to \mathsf{tree}(k_t, k_l) \\ \mathtt{node} \mapsto \forall k_t, k_l.(l : \mathsf{tree}(k_t, k_l); r : \mathsf{tree}(k_t, k_l); v : \mathsf{list}(k_l)), k_t \to \mathsf{tree}(k_t, k_l) \end{array}\right]$$

The k_t annotation is the per-tree-level amount of potential, and k_l the per-list-element potential. If we have the tree



[1] We retain variable names in the signature to reduce the quantity of notation; they are not strictly required.

then rather than looking at the total depth we need to consider the most *expensive* path in terms of potential by taking the maximum of

1. following the list at the root of the tree ($k_t + 4 \times k_l$);
2. following the left branch and the list there ($2 \times k_t + 0 \times k_l$);
3. following the right branch and the list there ($2 \times k_t + 2 \times k_l$).

An intuitive explanation is that the total potential assigned to the tree is the smallest amount of potential sufficient to recursively process the tree and lists using the full per-element amount of potential at each step.

6.2 Definition

The types are similar to Hofmann-Jost,

$$T := 1 \mid \mathsf{bool} \mid T \otimes T \mid (T, k_l) + (T, k_r) \mid ty(\overline{k}),$$

but the contexts are now structured as trees:

$$\Gamma := \cdot \mid x : T \mid \Gamma, \Gamma \mid \Gamma; \Gamma \mid k,$$

where $\cdot$ is the empty context and k is an annotation for a fixed amount of potential. The annotations are constraint variables as before, although we occasionally use an explicit 0. We take the two context formers, ‘,’ and ‘;’, to be associative throughout. As mentioned above, the ‘,’ means “sum the potential” and the ‘;’ means “take the maximum potential”. We will define this more precisely in the potential function for contexts, Υ_{c}. The function signatures take the form

$$\Gamma \to T, k \,|\, \Phi,$$

for a (tree-structured) context Γ, type T, annotation k and constraint set Φ. Similarly, constructor signatures have the form

$$\forall \overline{k}.\Gamma \to ty(\overline{k}),$$

where $\overline{k}$ is a sequence of annotations, as before. We require that variable names appear only once in the context Γ for both kinds of signature. We restrict the signatures for constructors in nullc to the form

$$\forall \overline{k}.k_i \to ty(\overline{k}).$$

Note that we can define algebraic datatypes which are the same except for the calculation of potential. For example, we can define a max-pair and a plus-pair for some types $ty_1(\overline{k_1})$ and $ty_2(\overline{k_2})$:

$$\Sigma(\mathsf{maxpair}) = \forall \overline{k_1} \cdot \overline{k_2}.\ ty_1(\overline{k_1});\ ty_2(\overline{k_2}) \to \mathsf{maxpair}(\overline{k_1} \cdot \overline{k_2}),$$
$$\Sigma(\mathsf{pluspair}) = \forall \overline{k_1} \cdot \overline{k_2}.\ ty_1(\overline{k_1}),\ ty_2(\overline{k_2}) \to \mathsf{pluspair}(\overline{k_1} \cdot \overline{k_2}),$$

(where $s_1 \cdot s_2$ is the concatenation of the sequences s_1 and s_2). The two types are the same operationally, except for the names of their constructors. However, for the first we will use the maximum potential of the two values in the pair as the pair's potential, and for the second we use the sum of each value's potential.

As we described above, we can define binary trees with potential proportional to their depth by taking the maximum potential of each pair of subtrees and adding some potential for the node:

$$\Sigma(\mathsf{leaf}) = \forall k_l, k_n.k_l \to \mathsf{booltree}(k_l, k_n),$$
$$\Sigma(\mathsf{node}) = \forall k_l, k_n.(x_l : \mathsf{booltree}(k_l, k_n); x_r : \mathsf{booltree}(k_l, k_n); x_v : \mathsf{bool}), k_n \to \mathsf{booltree}(k_l, k_n).$$

Like the nil annotations on lists in previous chapters, we can replace k_l with 0 in examples without much loss of generality:

$$\Sigma(\mathsf{leaf}) = \forall k_n.0 \to \mathsf{booltree}(k_n),$$
$$\Sigma(\mathsf{node}) = \forall k_n.(x_l : \mathsf{booltree}(k_n); x_r : \mathsf{booltree}(k_n); x_v : \mathsf{bool}), k_n \to \mathsf{booltree}(k_n).$$

In general we include one k per constructor, but this is not necessary for soundness. Note that we can still define a form of trees where the potential is proportional to the total size, by using the additive context former (,) throughout. (However, this does not entirely subsume the earlier analyses because the SHARE rule in the previous type systems has no complete replacement in this one, see the discussion on page 103 for more details.)

To define the potential functions and typing rules we need to be able to extract the names of the variables which appear in a context. Thus, we define a function to map a context to the sequence containing the variable names in order of appearance:

$$\begin{aligned} \mathrm{names}(x : T) &= (x), \\ \mathrm{names}(k) &= (), \\ \mathrm{names}(\Gamma, \Delta) &= \mathrm{names}(\Gamma) \cdot \mathrm{names}(\Delta), \\ \mathrm{names}(\Gamma; \Delta) &= \mathrm{names}(\Gamma) \cdot \mathrm{names}(\Delta). \end{aligned}$$

The potential functions can now be defined, using the contexts from the constructor signatures to calculate the potential of data structures, and plus and max for combining the potential of contexts depending upon the context former used. The functions for types and contexts are mutually recursive, so we distinguish between the two:

$$
\begin{aligned}
\Upsilon_{\mathsf{t}}(\sigma,*,1) = \Upsilon_{\mathsf{t}}(\sigma,\mathsf{true},\mathsf{bool}) &= \Upsilon_{\mathsf{t}}(\sigma,\mathsf{false},\mathsf{bool}) = 0 \\
\Upsilon_{\mathsf{t}}(\sigma,(v',v''),T'\otimes T'') &= \Upsilon_{\mathsf{t}}(\sigma,v',T') + \Upsilon_{\mathsf{t}}(\sigma,v'',T''), \\
\Upsilon_{\mathsf{t}}(\sigma,\mathsf{inl}(v),(T',k')+(T'',k'')) &= k' + \Upsilon_{\mathsf{t}}(\sigma,v,T'), \\
\Upsilon_{\mathsf{t}}(\sigma,\mathsf{inr}(v),(T',k')+(T'',k'')) &= k'' + \Upsilon_{\mathsf{t}}(\sigma,v,T''), \\
\Upsilon_{\mathsf{t}}(\sigma,\mathsf{null},ty(\bar{k})) &= k_i \quad \text{where} \quad c \in \mathsf{nullc} \\
&\qquad\qquad \text{and} \quad \Sigma(c)[\bar{k}] = k_i \to ty(\bar{k}), \\
\Upsilon_{\mathsf{t}}(\sigma,l,ty(\bar{k})) &= \Upsilon_{\mathsf{c}}(\sigma\setminus l,[x_1\mapsto v_1,\ldots,x_p\mapsto v_p],\Gamma), \\
&\qquad \text{where} \quad \sigma(l) = (c,v_1,\ldots,v_p), \\
&\qquad\quad \text{and} \quad \Sigma(c)[\bar{k}] = \Gamma \to ty(\bar{k}),\ (x_1,\ldots,x_p) = \mathrm{names}(\Gamma),
\end{aligned}
$$

$$
\begin{aligned}
\Upsilon_{\mathsf{c}}(\sigma,S,\cdot) &= 0, \\
\Upsilon_{\mathsf{c}}(\sigma,S,x:T) &= \Upsilon_{\mathsf{t}}(\sigma,S(x),T), \\
\Upsilon_{\mathsf{c}}(\sigma,S,k) &= k, \\
\Upsilon_{\mathsf{c}}(\sigma,S,(\Gamma,\Delta)) &= \Upsilon_{\mathsf{c}}(\sigma,S,\Gamma) + \Upsilon_{\mathsf{c}}(\sigma,S,\Delta), \\
\Upsilon_{\mathsf{c}}(\sigma,S,(\Gamma;\Delta)) &= \max\{\Upsilon_{\mathsf{c}}(\sigma,S,\Gamma),\Upsilon_{\mathsf{c}}(\sigma,S,\Delta)\}.
\end{aligned}
$$

The typing rules for expressions are presented in Figures 6.1 and 6.2. The judgements take the form $\Gamma \vdash^{g,t}_{\Sigma,F} e : T,k \,|\, \Phi$ where Γ is a tree-structured context as defined above, Σ contains the signatures, F is the set of previously defined functions, g is the name of the current function and t the tail position flag. The expression has type T and a fixed amount of potential k is added to the potential from T. As before, Φ is a set of constraints on annotations. The notation $\Gamma()$ represents a context with a 'hole' in place of a subcontext, and $\Gamma(\Delta)$ is $\Gamma()$ with the hole replaced by Δ.

The constraints are still linear equalities and inequalities in terms of the annota-

$$\frac{}{k \vdash^{g,t}_{\Sigma,F} * : 1, k' \,|\, \{k \geq k'\}} \quad \text{(D-UNIT)}$$

$$\frac{c \in \{\mathsf{true}, \mathsf{false}\}}{k \vdash^{g,t}_{\Sigma,F} c : \mathsf{bool}, k' \,|\, \{k \geq k'\}} \quad \text{(D-BOOL)}$$

$$\frac{}{x : T, k \vdash^{g,t}_{\Sigma,F} x : T, k' \,|\, \{k \geq k'\}} \quad \text{(D-VAR)}$$

$$\frac{\begin{array}{c} f \in F \qquad \Sigma(f) = \Gamma \to T, k_1' | \Phi' \qquad (y_1, \ldots, y_p) = \mathrm{names}(\Gamma) \\ \Phi = \rho(\Phi') \cup \{k \geq \mathrm{stack}'(g, f, t),\ k + \rho(k_1') \geq k'\} \end{array}}{\rho(\Gamma)[x_1/y_1, \ldots, x_p/y_p], k \vdash^{g,t}_{\Sigma,F} f(x_1, \ldots, x_p) : \rho(T), k' \,|\, \Phi} \quad \text{(D-FUN)}$$

$$\frac{\begin{array}{c} f \notin F \qquad \Sigma(f) = \Gamma \to T, k_1' | \Phi' \qquad (y_1, \ldots, y_p) = \mathrm{names}(\Gamma) \\ \Phi = \{k \geq \mathrm{stack}'(g, f, t),\ k + k_1' \geq k'\} \end{array}}{\Gamma[x_1/y_1, \ldots, x_p/y_p], k \vdash^{g,t}_{\Sigma,F} f(x_1, \ldots, x_p) : T, k' \,|\, \Phi} \quad \text{(D-FUNDEF)}$$

$$\frac{\Delta \vdash^{g,\mathsf{false}}_{\Sigma,F} e_1 : T_0, k_0 \,|\, \Phi_1 \qquad \Gamma(x : T_0, k_0) \vdash^{g,t}_{\Sigma,F} e_2 : T, k' \,|\, \Phi_2}{\Gamma(\Delta) \vdash^{g,t}_{\Sigma,F} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T, k' \,|\, \Phi_1 \cup \Phi_2} \quad \text{(D-LET)}$$

$$\frac{\Gamma(\cdot) \vdash^{g,t}_{\Sigma,F} e_1 : T, k \,|\, \Phi_1 \qquad \Gamma(\cdot) \vdash^{g,t}_{\Sigma,F} e_2 : T, k \,|\, \Phi_2}{\Gamma(x : \mathsf{bool}) \vdash^{g,t}_{\Sigma,F} \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T, k \,|\, \Phi_1 \cup \Phi_2} \quad \text{(D-IF)}$$

$$\frac{}{x_1 : T_1, x_2 : T_2, k \vdash^{g,t}_{\Sigma,F} (x_1, x_2) : T_1 \otimes T_2, k' \,|\, \{k \geq k'\}} \quad \text{(D-PAIR)}$$

$$\frac{\Gamma(x_1 : T_1, x_2 : T_2) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi}{\Gamma(x : T_1 \otimes T_2) \vdash^{g,t}_{\Sigma,F} \mathsf{match}\ x\ \mathsf{with}\ (x_1, x_2) \to e : T, k' \,|\, \Phi} \quad \text{(D-PAIRELIM)}$$

$$\frac{}{x : T_1, k_1, k \vdash^{g,t}_{\Sigma,F} \mathsf{inl}(x) : (T_1, k_1) + (T_2, k_2), k' \,|\, \{k \geq k'\}} \quad \text{(D-INL)}$$

$$\frac{}{x : T_2, k_2, k \vdash^{g,t}_{\Sigma,F} \mathsf{inr}(x) : (T_1, k_1) + (T_2, k_2), k' \,|\, \{k \geq k'\}} \quad \text{(D-INR)}$$

$$\frac{\Gamma(x_1 : T_1, k_1) \vdash^{g,t}_{\Sigma,F} e_1 : T, k' \,|\, \Phi_1 \qquad \Gamma(x_2 : T_2, k_2) \vdash^{g,t}_{\Sigma,F} e_2 : T, k' \,|\, \Phi_2}{\Gamma(x : (T_1, k_1) + (T_2, k_2)) \vdash^{g,t}_{\Sigma,F} \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_1) \to e_1 \mid \mathsf{inr}(x_2) \to e_2 : T, k' \,|\, \Phi_1 \cup \Phi_2} \quad \text{(D-SUMELIM)}$$

Figure 6.1: Typing rules for expressions in the depth analysis

$$\frac{\Sigma(c)[\overline{k}] = \Gamma \to ty(\overline{k}) \qquad (y_1,\ldots,y_p) = \text{names}(\Gamma)}{\Gamma[x_1/y_1,\ldots,x_p/y_p], k \vdash^{g,t}_{\Sigma,F} c(x_1,\ldots,x_p) : ty(\overline{k}), k' \,|\, \{k \geq k'\}} \text{ (D-CONSTRUCT)}$$

$$\frac{\text{for all } i,\ 1 \leq i \leq m, \quad \Gamma() \mathbin{\triangleright} ty(\overline{k}) \vdash^{g,t}_{\Sigma,F} p_i \to e_i : T, k' \,|\, \Phi_i}{\Gamma(x : ty(\overline{k})) \vdash^{g,t}_{\Sigma,F} \text{match } x \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m : T, k' \,|\, \bigcup_i \Phi_i} \text{ (D-MATCH)}$$

$$\frac{\Gamma(\Delta[x_1/y_1,\ldots,x_p/y_p]) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \qquad \Sigma(c)[\overline{k}] = \Delta \to ty(\overline{k}) \qquad (y_1,\ldots,y_p) = \text{names}(\Delta)}{\Gamma() \mathbin{\triangleright} ty(\overline{k}) \vdash^{g,t}_{\Sigma,F} c(x_1,\ldots,x_p)\langle' \rangle \to e : T, k' \,|\, \Phi} \text{ (D-CASE)}$$

$$\frac{\Gamma(\Delta) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi}{\Gamma(\Gamma'(\Delta)) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi} \text{ (D-WEAKEN)}$$

$$\frac{\Gamma(x : T[k_1/k]) \vdash^{g,t}_{\Sigma,F} e : T', k' \,|\, \Phi}{\Gamma(x : T) \vdash^{g,t}_{\Sigma,F} e : T', k' \,|\, \Phi \cup \{k \geq k_1\}} \text{ (D-WEAKENA)}$$

$$\frac{\Gamma(\Delta') \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \qquad \Delta \cong \Delta' \,|\, \Phi'}{\Gamma(\Delta) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \cup \Phi'} \text{ (D-}\equiv\text{)}$$

$$\frac{\begin{array}{c} q\Delta = \Delta_q \,|\, \Phi_q \qquad (1-q)\Delta' = \Delta'_q \,|\, \Phi'_q \\ \Gamma(\Delta_q, \Delta'_q) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \end{array}}{\Gamma(\Delta;\Delta') \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \cup \Phi_q \cup \Phi'_q \cup \{0 \leq q, q \leq 1\}} \text{ (D-SPLIT)}$$

Figure 6.2: Typing rules for expressions in the depth analysis (continued)

$$\Gamma, \Delta \cong \Delta, \Gamma \,|\, \emptyset \text{ (plus-commute)} \qquad \Gamma;\Delta \cong \Delta;\Gamma \,|\, \emptyset \text{ (max-commute)}$$

$$\Gamma, (\Delta;\Delta') \cong (\Gamma,\Delta);(\Gamma,\Delta') \,|\, \emptyset \text{ (distribute)} \qquad \Gamma \cong \Gamma;\Gamma \,|\, \emptyset \text{ (max-contract)}$$

$$\Gamma \cong \Gamma, \cdot \,|\, \emptyset \text{ (plus-empty)} \qquad \Gamma \cong \Gamma;\cdot \,|\, \emptyset \text{ (max-empty)}$$

$$\Gamma \cong \Gamma, 0 \,|\, \emptyset \text{ (plus-zero)} \qquad \Gamma \cong \Gamma;0 \,|\, \emptyset \text{ (max-zero)}$$

$$\frac{q\Gamma = \Gamma_q \,|\, \Phi_q \qquad (1-q)\Gamma = \Gamma'_q \,|\, \Phi'_q}{\Gamma \cong \Gamma_q, \Gamma'_q \,|\, \Phi_q \cup \Phi'_q \cup \{0 \leq q, q \leq 1\}} \text{ (plus-contract)} \qquad \frac{\Gamma \cong \Delta \,|\, \Phi}{\Delta \cong \Gamma \,|\, \Phi} \text{ (symmetry)}$$

Figure 6.3: Equivalent contexts (for the D-$\equiv$ rule)

$$\frac{}{q(1) = 1 \,|\, \emptyset} \qquad \frac{}{q(\mathsf{bool}) = \mathsf{bool} \,|\, \emptyset}$$

$$\frac{q(T_1) = T_1' \,|\, \Phi_1 \qquad q(T_2) = T_2' \,|\, \Phi_2}{q(T_1 \otimes T_2) = T_1' \otimes T_2' \,|\, \Phi_1 \cup \Phi_2}$$

$$\frac{q(T_1) = T_1' \,|\, \Phi_1 \qquad q(T_2) = T_2' \,|\, \Phi_2}{q((T_1, k_1) + (T_2, k_2)) = (T_1', k_1') + (T_2', k_2') \,|\, \Phi_1 \cup \Phi_2 \cup \{qk_1 = k_1', qk_2 = k_2'\}}$$

$$\frac{}{q(ty(\overline{k})) = ty(\overline{k'}) \,|\, \{qk_i = k_i' : \forall i\}}$$

$$\frac{}{q(\cdot) = \cdot \,|\, \emptyset} \qquad \frac{q(T) = T' \,|\, \Phi}{q(x : T) = x : T' \,|\, \Phi} \qquad \frac{}{q(k) = k' \,|\, \{qk = k'\}}$$

$$\frac{q(\Gamma_1) = \Gamma_1' \,|\, \Phi_1 \qquad q(\Gamma_2) = \Gamma_2' \,|\, \Phi_2}{q(\Gamma_1, \Gamma_2) = \Gamma_1', \Gamma_2' \,|\, \Phi_1 \cup \Phi_2} \qquad \frac{q(\Gamma_1) = \Gamma_1' \,|\, \Phi_1 \qquad q(\Gamma_2) = \Gamma_2' \,|\, \Phi_2}{q(\Gamma_1; \Gamma_2) = \Gamma_1'; \Gamma_2' \,|\, \Phi_1 \cup \Phi_2}$$

Figure 6.4: Rules for scaling annotations

tions, but we allow extra constraint variables in the form of a scaling factor q:

$$\begin{aligned} a_1k_1 + \cdots + a_nk_n &= a_{n+1}k_{n+1} + \cdots + a_mk_m + c, \\ a_1k_1 + \cdots + a_nk_n &\geq a_{n+1}k_{n+1} + \cdots + a_mk_m + c, \text{ or} \\ qk_1 &= k_2, \end{aligned}$$

where $a_i \in \mathbb{Q}$, $c \in \mathbb{Q}$ and q is a constraint variable for a rational value (constrained to be in the range 0 to 1). We will discuss the need for these scaling variables shortly, and then how to construct linear programs without them in Section 6.4.

The function rules D-FUN and D-FUNDEF, and the algebraic datatype rules D-CONSTRUCT and D-CASE use the contexts from the signatures to form the typing contexts used in the appropriate judgements, renaming the variables and annotations as necessary. Note that the judgements for D-CASE use a context-with-a-hole $\Gamma()$ so that the variable's contents are placed in the correct position of the resulting context. The $\langle' \rangle$ notation means that the rule can be used for cases with and without a $'$ — heap deallocation makes no difference to the stack space available.

The D-LET rule uses some subcontext Δ for the typing of e_1, and replaces it with $x : T_0, k_0$ in the typing of e_2. This replacement relies upon the stack discipline for soundness: the operational semantics guarantees that the stack memory available for evaluating e_2 is the same as for the entire let expression, and the potential for $x : T_0, k_0$ is at most the potential of Δ because no extra stack memory can be freed by the evaluation

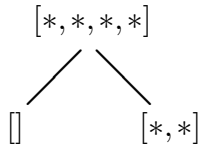
of e_1. Thus if $\Gamma(x : T_0, k_0)$ has enough potential for e_2 then $\Gamma(\Delta)$ has enough potential for the entire let.

However, if we were considering heap allocation then there may not be enough free memory after evaluating e_1 for the potential of $\Gamma(x : T_0, k_0)$ to be a lower bound on the amount left. For example, if we have $\Delta = a : \mathsf{intlist}(1)$ and $\Gamma(\Delta) = a : \mathsf{intlist}(1);\ b : \mathsf{intlist}(1)$ then the initial potential is $\max\{|a|, |b|\}$. Now suppose that e_1 allocates $|a|$ units of space, leaving the resulting $x : T_0, k_0$ with zero units of potential. However, $\Gamma(x : T_0, k_0)$ still has $\max\{0, |b|\} = |b|$ units of potential, even though there may be no free memory left. For a similar analysis to bound heap space we would need to change $\Gamma()$ to remove this excess potential when typing e_2. We will briefly discuss this in Section 9.2.3.

We move on to the four structural typing rules. The D-WEAKEN rule removes unnecessary parts of the context and D-WEAKENA replaces any annotation in the context with a lower one. The principle use of D-WEAKENA is to lower the potential in one branch of computation so that it provides a result of the same type as other branches. The D-$\equiv$ rule allows a number of bidirectional context transformations to be used. These are given in Figure 6.3. The transformations provide commutativity and contraction for both context forms, distribution, and the introduction and removal of empty contexts and zero units of fixed potential. The intuitive idea behind these transformations is that they all preserve the potential of the context.

Finally, D-SPLIT allows a maximum (;) context to be turned into a plus (,) context by taking a fraction of the potential of each subcontext. Both D-$\equiv$ and D-SPLIT use an auxiliary set of rules in Figure 6.4 to take a 'fraction' of the potential of a context.

Note that the plus-contraction form of D-$\equiv$, $\Gamma \cong \Gamma_q, \Gamma'_q \,|\, \ldots$, is more constrained than its counterpart in Hofmann-Jost, SHARE. The SHARE rule allows each annotation in a type to be split up independently, whereas the plus-contraction here only allows uniform splitting by some fraction, q. To understand this restriction, consider the tree of unit lists from before:



Suppose we have a per-tree-node potential $k_t = 10$ and a per-list-element potential of $k_l = 1$. The potential of the entire structure is the maximum potential path from tree root to list end. Here that path goes to the right subtree and follows the list, with a total

potential of $1 \times k_t + 2 \times k_l = 12$. If we were allowed to split the annotations separately we could construct two variables with the following potential:

Total: $k_t + 2 \times k_l = 10$. Total: $0 \times k_t + 4 \times k_l = 4$.

Together, we would have an *increased* potential of 14, because splitting the annotations independently has changed which parts of the data structure determine the potential. In general, the best we can do while guaranteeing that the overall potential remains unchanged is to scale the annotations uniformly. That is the approach we take here.

As with our previous type systems, we provide rules to define when an entire program is well-typed. These are presented in Figure 6.5, and are essentially the same as before.

$$\frac{\Sigma(f) = \Gamma \to T, k_1' | \Phi \qquad (x_1, \ldots, x_p) = \text{names}(\Gamma) \qquad \Gamma \vdash^{f,\mathsf{true}}_{\Sigma,F} e_f : T, k_1' \,|\, \Phi'}{\vdash_{\Sigma,F} f(x_1, \ldots, x_p) = e_f \Rightarrow \{f\}, \Phi'}$$

$$\frac{\vdash_{\Sigma,F} D \Rightarrow F', \Phi' \qquad \vdash_{\Sigma,F} B \Rightarrow F'', \Phi''}{\vdash_{\Sigma,F} D \text{ and } B \Rightarrow F' \cup F'', \Phi' \cup \Phi''}$$

$$\frac{\vdash_{\Sigma,F} B \Rightarrow F', \Phi' \qquad \forall f \in F'. \quad \Sigma(f) = \ldots | \Phi'}{\vdash_{\Sigma,F} \mathsf{let}\ B} \qquad \frac{\vdash_{\Sigma,F} B \Rightarrow F', \Phi' \qquad \vdash_{\Sigma,F \cup F'} P \qquad \forall f \in F'. \quad \Sigma(f) = \ldots | \Phi'}{\vdash_{\Sigma,F} \mathsf{let}\ B\ P}$$

Figure 6.5: Typing rules for function signatures in the depth analysis

6.3 Soundness

The main theorem will show that the bound on the stack memory requirements for an expression predicted by the type system is sufficient for execution and that the potential of the result is consistent with that amount of stack memory. The proof proceeds by showing that potential reflects the available stack memory throughout the execution. Before that we prove several lemmas to show that changes to contexts have the expected effect on the potential, starting with subcontext replacement:

Lemma 6.1. *Suppose Δ and Δ' are the subcontexts in some $\Gamma(\Delta)$ and $\Gamma(\Delta')$, and that there is a store σ and environment S such that Υ_c is well defined on Δ, Δ' and $\Gamma(\Delta)$. Then*

1. *if we have $\Upsilon_c(\sigma, S, \Delta) = \Upsilon_c(\sigma, S, \Delta')$ then $\Upsilon_c(\sigma, S, \Gamma(\Delta'))$ is well defined and $\Upsilon_c(\sigma, S, \Gamma(\Delta)) = \Upsilon_c(\sigma, S, \Gamma(\Delta'))$, and*
2. *if we have $\Upsilon_c(\sigma, S, \Delta) \geq \Upsilon_c(\sigma, S, \Delta')$ then $\Upsilon_c(\sigma, S, \Gamma(\Delta'))$ is well defined and $\Upsilon_c(\sigma, S, \Gamma(\Delta)) \geq \Upsilon_c(\sigma, S, \Gamma(\Delta'))$.*

Proof. We use induction on the structure of $\Gamma()$. For the base case we use the premise about Δ. The inductive step cases are formed using ',' and ';', which add and take the maximum potential respectively. Both preserve equality and inequality, yielding the result. □

We give three lemmas on weakening, one for contexts, one for annotations and one for unused variables:

Lemma 6.2. *For any $\Gamma(), \Delta, \sigma, S$, if $\Upsilon_c(\sigma, S, \Gamma(\Delta))$ is defined then*

$$\Upsilon_c(\sigma, S, \Gamma(\Delta)) \geq \Upsilon_c(\sigma, S, \Delta).$$

Proof. By induction on $\Gamma()$, using the monotonicity of $+$ and max. □

Lemma 6.3. *For any assignment of rationals to constraint variables, if $\Upsilon_t(\sigma, v, T)$ is defined and $k \geq k_1$ then we have*

$$\Upsilon_t(\sigma, v, T) \geq \Upsilon_t(\sigma, v, T[k_1/k]).$$

Proof. Straightforward induction on the definition of $\Upsilon_t(\sigma, v, T)$. At each point the subcases are combined by max or +, so replacing k by k' cannot increase the overall value. □

Lemma 6.4. *Given a derivation of*

$$\Gamma \vdash^{g,t}_{\Sigma,F} e : T, k \,|\, \Phi$$

then for any $x \notin \mathrm{FV}(e)$ there is also a derivation of

$$\Gamma' \vdash^{g,t}_{\Sigma,F} e : T, k \,|\, \Phi'$$

for some $\Phi' \subseteq \Phi$, where Γ' is Γ with every occurrence of x replaced by the empty context.

Proof. By induction on the structure of the derivation. As $x \notin \mathrm{FV}(e)$ the only rules which may directly involve a $x : T$ subcontext are D-SPLIT, which will just leave the empty context unchanged and so requires one less constraint, and D-$\equiv$'s contraction equivalences, where the replacement of x by the empty context will merely remove the relevant constraints. □

Now we show that the scaling relation defined in Figure 6.4 has the desired effect on the potential.

Lemma 6.5. *If $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ is defined for some σ, S and Γ and we have $q(\Gamma) = \Gamma_q \,|\, \Phi$ then for any assignment satisfying Φ we have*

$$q\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma) = \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma_q).$$

Proof. Straightforward induction on the derivation of $q(\Gamma) = \Gamma_q \,|\, \Phi$. Essentially, $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ is made up of annotations combined by $+$ and max. In Γ_q all of the annotations are scaled by q and so the q can be factored out. □

The bidirectional transformations for the D-$\equiv$ preserve the potential of the context:

Lemma 6.6. *If we have two contexts Γ and Γ' with a store σ and environment S such that $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ is well-defined and $\Gamma \cong \Gamma'$, then $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma) = \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma')$.*

Proof. We consider each case:

$\Gamma, \Delta \cong \Delta, \Gamma$: From the definition of Υ_{c} using the commutativity of $+$,

$$\begin{aligned}\Upsilon_{\mathsf{c}}(\sigma, S, (\Gamma, \Delta)) &= \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma) + \Upsilon_{\mathsf{c}}(\sigma, S, \Delta) \\ &= \Upsilon_{\mathsf{c}}(\sigma, S, \Delta) + \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma) = \Upsilon_{\mathsf{c}}(\sigma, S, (\Delta, \Gamma)).\end{aligned}$$

$\Gamma; \Delta \cong \Gamma; \Delta$: From the definition again, using the commutativity of max.

$\Gamma, (\Delta; \Delta') \cong (\Gamma, \Delta); (\Gamma, \Delta')$: From the definition of Υ_{c}, using the distributivity of $+$ over max.

$\Gamma \cong \Gamma; \Gamma$: From the idempotency of max.

$\Gamma \cong \Gamma, \cdot$, $\Gamma \cong \Gamma; \cdot$, $\Gamma \cong \Gamma, 0$, $\Gamma \cong \Gamma; 0$: Follows immediately from the definition of Υ_{c} on each right hand side.

$\Gamma \cong \Gamma_q, \Gamma'_q$: Follows from Lemma 6.5.

$\Gamma \cong \Delta \Rightarrow \Delta \cong \Gamma$: Follows by the relevant earlier case and the symmetry of $=$. □

Now we can state the main theorem to show that the potential provides a bound on the stack space used when evaluating an expression.

Theorem 6.7. *Let* $\mathrm{size}(c) = 0$ *for all constructors c. If an expression e in some well-typed program has a typing*

$$\Gamma \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi$$

with an assignment of nonnegative rationals to constraint variables which satisfies the constraints in g's signature, and an evaluation $S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma'$ *satisfying the benign sharing conditions, and* $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ *is defined, then for any* $q \in \mathbb{Q}^+$ *and* $m \in \mathbb{N}$ *such that*

$$m \geq \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma) + q$$

m will be a sufficient amount of stack space for the execution to succeed,

$$m, S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma', m,$$

and

$$m \geq \Upsilon_{\mathsf{t}}(\sigma', v, T) + k' + q.$$

Proof. We proceed by simultaneous induction on the evaluation and the typing derivations. First, note that whenever we use a value from S or σ we can be sure that it has the expected form for its type because otherwise $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ would not be defined.

For the leaf evaluation rules, no extra stack memory is required so the execution will always succeed. Note that all of the corresponding typing rules have some annotation k in the context with the constraint that $k \geq k'$. To obtain $m \geq \Upsilon_{\mathsf{t}}(\sigma', v, T) + k' + q$ it suffices to show that $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma) \geq \Upsilon_{\mathsf{t}}(\sigma', v, T) + k'$:

D-BOOL. Immediate from $\Upsilon_{\mathsf{t}}(\sigma, c, \mathsf{bool}) = 0$ and $\Gamma = k$.

D-VAR. The context has the form $x : T, k$, and by the definition of Υ_{c}

$$\Upsilon_{\mathsf{c}}(\sigma, S, (x : T, k)) \geq \Upsilon_{\mathsf{t}}(\sigma, S(x), T) + k'.$$

D-PAIR. The context has the form $x_1 : T_1, x_2 : T_2, k$, and

$$\begin{aligned} \Upsilon_{\mathsf{c}}(\sigma, S, (x_1 : T_1, x_2 : T_2, k)) &= \Upsilon_{\mathsf{t}}(\sigma, S(x_1), T_1) + \Upsilon_{\mathsf{t}}(\sigma, S(x_2), T_2) + k \\ &\geq \Upsilon_{\mathsf{t}}(\sigma, (S(x_1), S(x_2)), T_1 \otimes T_2) + k'. \end{aligned}$$

D-INL. We have the context $x : T_1, k_1, k$, and

$$\begin{aligned}\Upsilon_{\mathsf{c}}(\sigma, S, (x : T_1, k_1, k)) &= \Upsilon_{\mathsf{t}}(\sigma, S(x), T_1) + k_1 + k \\ &\geq \Upsilon_{\mathsf{t}}(\sigma, \mathsf{inl}(S(x)), (T_1, k_1) + (T_2, k_2)) + k'.\end{aligned}$$

D-INR. Analogous to D-INL.

D-CONSTRUCT with E-CONSTRUCTN. The constructor is some nullary $c \in \mathsf{nullc}$ with a signature of the form $\Sigma(c)[\overline{k}] = k_i \to ty(\overline{k})$. Recall that such constructors are unique for their type, ty. The context must have the form k_i, k, and so the result follows from the definition of Υ_{c}:

$$\Upsilon_{\mathsf{t}}(\sigma, \mathsf{null}, ty(\overline{k})) = k_i.$$

D-CONSTRUCT with E-CONSTRUCT. The context has the form $\Gamma[x_1/y_1, \ldots, x_p/y_p], k$ where $(y_1, \ldots, y_p) = \mathrm{names}(\Gamma)$, and the constructor c has a signature $\Sigma(c)[\overline{k}] = \Gamma \to ty(\overline{k})$. Now,

$$\begin{aligned}\Upsilon_{\mathsf{c}}(\sigma, S, (\Gamma[x_1/y_1, \ldots, x_p/y_p], k)) &= \Upsilon_{\mathsf{c}}(\sigma, [y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p)], \Gamma) + k \\ &\geq \Upsilon_{\mathsf{t}}(\sigma[l \mapsto s], l, ty(\overline{k})) + k'\end{aligned}$$

as required, where $s = (c, S(x_1), \ldots, S(x_p))$.

The other rules need to use the induction hypothesis. The precondition on m can be satisfied by showing that the original $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ is larger than or equal to its counterpart for the induction hypothesis. The result of the induction hypothesis is sufficient for most of the rules, where the resulting value and type from the induction hypothesis are also the value and type of the current expression. The D-FUN rule is a little different due to the stack space used, and D-LET rule uses the induction hypothesis twice, see below.

We start with the typing rules which have no operational effect.

D-WEAKEN. By Lemma 6.2, $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma'(\Delta)) \geq \Upsilon_{\mathsf{c}}(\sigma, S, \Delta)$ and then applying Lemma 6.1 we obtain

$$\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(\Gamma'(\Delta))) \geq \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(\Delta)).$$

It is then sufficient to apply the induction hypothesis.

D-WEAKENA. Combining Lemma 6.3 and Lemma 6.1 we have

$$\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(x:T)) \geq \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(x:T[k_1/k]))$$

and so can apply the induction hypothesis to obtain the result.

D-$\equiv$. By Lemma 6.6, $\Upsilon_{\mathsf{c}}(\sigma, S, \Delta) = \Upsilon_{\mathsf{c}}(\sigma, S, \Delta')$, and so by Lemma 6.1, $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(\Delta)) = \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(\Delta'))$. The result follows from the induction hypothesis.

D-SPLIT. As $q \in [0,1]$,

$$\max\{\Upsilon_{\mathsf{c}}(\sigma, S, \Delta), \Upsilon_{\mathsf{c}}(\sigma, S, \Delta')\} \geq q\Upsilon_{\mathsf{c}}(\sigma, S, \Delta) + (1-q)\Upsilon_{\mathsf{c}}(\sigma, S, \Delta').$$

From this and Lemma 6.5 we get

$$\Upsilon_{\mathsf{c}}(\sigma, S, (\Delta; \Delta')) \geq \Upsilon_{\mathsf{c}}(\sigma, S, (q\Delta, (1-q)\Delta'))$$

and we can apply the induction hypothesis.

D-LET with E-LET-TAIL. First, consider the induction hypothesis on e_1: for any $q \in \mathbb{Q}^+$ and $m_1 \in \mathbb{N}$ such that $m_1 \geq \Upsilon_{\mathsf{c}}(\sigma, S, \Delta) + q$, we have $m_1 \geq \Upsilon_{\mathsf{t}}(\sigma_0, v_0, T_0) + k_0 + q$. As $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(\Delta)) \geq \Upsilon_{\mathsf{c}}(\sigma, S, \Delta)$ by Lemma 6.2, we can take $m_1 = m$ and so the execution of e_1 will succeed.

We can also use the induction hypothesis to deduce that the potential has not increased. If we set $m_1 = \lceil \Upsilon_{\mathsf{c}}(\sigma, S, \Delta) \rceil$ and $q = m_1 - \Upsilon_{\mathsf{c}}(\sigma, S, \Delta)$ we can see that $m_1 - q = \Upsilon_{\mathsf{c}}(\sigma, S, \Delta)$, and from the induction hypothesis we know that $m_1 - q \geq \Upsilon_{\mathsf{t}}(\sigma_0, v_0, T_0) + k_0$. Thus,

$$\Upsilon_{\mathsf{c}}(\sigma, S, \Delta) \geq \Upsilon_{\mathsf{t}}(\sigma_0, v_0, T_0) + k_0.$$

By Lemma 6.4 we can replace the variables in $\Gamma()$ which are not used in e_2 by the empty context to yield some $\Gamma'()$, and still obtain a typing for e_2 under the same constraints. Now let $d = \Upsilon_{\mathsf{c}}(\sigma, S, \Delta)$. Thus, we can use Lemma 6.1 to replace Δ,

$$\begin{aligned}
\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(\Delta)) &= \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(d)) \\
&\geq \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma'(d)) \\
&= \Upsilon_{\mathsf{c}}(\sigma_0, S, \Gamma'(d)) \\
&\geq \Upsilon_{\mathsf{c}}(\sigma_0, S[x \mapsto v_0], \Gamma'(x : T_0, k_0)).
\end{aligned}$$

where the change from σ to σ_0 is justified by the restriction of Γ to free variables and the benign sharing condition (from page 10)

$$\sigma \restriction \mathcal{R}(\sigma, S_{e_2}) = \sigma_0 \restriction \mathcal{R}(\sigma, S_{e_2}), \tag{2.2}$$

where $S_{e_2} = S \restriction (\mathrm{FV}(e_2) \setminus x)$. Finally, we apply the induction hypothesis to e_2 to obtain the result.

D-FUN. There must be a typing

$$\Gamma \vdash^{f,\mathsf{true}}_{\Sigma,F} e_f : T, k_1' \,|\, \Phi'$$

where e_f is the body of the function f, because the entire program is well typed. We can apply the substitution of annotations, ρ, to its derivation to obtain

$$\rho(\Gamma) \vdash^{f,\mathsf{true}}_{\Sigma,F} e_f : \rho(T), \rho(k_1') \,|\, \rho(\Phi').$$

Now,

$$\begin{aligned} m &\geq \Upsilon_{\mathsf{c}}(\sigma, S, (\rho(\Gamma[x_1/y_1, \ldots, x_p/y_p], k)) + q \\ &= \Upsilon_{\mathsf{c}}(\sigma, [y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p)], \rho(\Gamma)) + k + q \\ &= \Upsilon_{\mathsf{c}}(\sigma, [y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p)], \rho(\Gamma)) + \mathrm{stack}'(g, f, t) + (q + k - \mathrm{stack}'(g, f, t)). \end{aligned}$$

The constraint on k guarantees that $q + k - \mathrm{stack}'(g, f, t) \geq 0$ so we can apply the induction hypothesis on e_f with that as the constant (that is, the 'q' for e_f). From the induction hypothesis we can conclude

$$\begin{aligned} m &\geq \Upsilon_{\mathsf{t}}(\sigma', v, \rho(T)) + \rho(k_1') + (q + k - \mathrm{stack}'(g, f, t)) + \mathrm{stack}'(g, f, t) \\ &\geq \Upsilon_{\mathsf{t}}(\sigma', v, \rho(T)) + k'. \end{aligned}$$

as required, using the constraint on $k + \rho(k_1')$.

D-FUNDEF. As D-FUN with the identity in place of ρ. The constraints in Φ' will form part of the current function's signature, and so will be satisfied.

D-IF with EIFTRUE or EIFFALSE. From the definition, $\Upsilon_{\mathsf{t}}(\sigma, c, \mathsf{bool}) = 0 = \Upsilon_{\mathsf{c}}(\sigma, S, \cdot)$, and using Lemma 6.1, $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(x : \mathsf{bool})) = \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(\cdot))$. Hence we can use the induction hypothesis for the appropriate branch.

D-PAIRELIM. By definition

$$\Upsilon_{\mathsf{c}}(\sigma, S(x), x : T_1 \otimes T_2) = \Upsilon_{\mathsf{t}}(\sigma, v_1, T_1) + \Upsilon_{\mathsf{t}}(\sigma, v_2, T_2),$$

and so by Lemma 6.1 we can apply the induction hypothesis on e.

D-SUMELIM with E-MATCHINL. From the definition

$$\begin{aligned}\Upsilon_{\mathsf{t}}(\sigma, S(x), (T_1,k_1)+(T_2,k_2)) &= \Upsilon_{\mathsf{t}}(\sigma, v, T_1)+k_1 \\ &= \Upsilon_{\mathsf{c}}(\sigma, S[x_1 \mapsto v], (x_1 : T_1, k_1)),\end{aligned}$$

where $S(x) = \mathsf{inl}(v)$ for some value v. We can apply the induction hypothesis to e_1 after using Lemma 6.1 to account for the rest of the context.

D-SUMELIM with E-MATCHINR. Analogous to the E-MATCHINL case.

D-MATCH and D-CASE with E-MATCHN. From the evaluation $S(x) = \mathsf{null}$ and $c \in \mathsf{nullc}$. Thus there is a signature of the form $\Sigma(c)[\overline{k}] = k_i \to ty(\overline{k})$ because c is the unique constructor in nullc for ty. By the definition of Υ_{c},

$$\Upsilon_{\mathsf{c}}(\sigma, S, x : ty(\overline{k})) = \Upsilon_{\mathsf{t}}(\sigma, \mathsf{null}, ty(\overline{k})) = k_i = \Upsilon_{\mathsf{c}}(\sigma, S, k_i).$$

Lemma 6.1 and the induction hypothesis can now be used as usual.

D-MATCH and D-CASE with E-MATCH. From the evaluation $S(x) = l$ for some $l \in \mathsf{loc}$ and $\sigma(l) = (c, v_1, \ldots, v_p)$. From the typing rules $\Sigma(c)[\overline{k}] = \Delta \to ty(\overline{k})$. Using the definition of Υ_{c},

$$\begin{aligned}\Upsilon_{\mathsf{c}}(\sigma, S, x : ty(\overline{k})) = \Upsilon_{\mathsf{t}}(\sigma, l, ty(\overline{k})) &= \Upsilon_{\mathsf{c}}(\sigma \setminus l, [x_1 \mapsto v_1, \ldots, x_p \mapsto v_p], \Delta) \\ &= \Upsilon_{\mathsf{c}}(\sigma \setminus l, [y_1 \mapsto v_1, \ldots, x_p \mapsto v_p], \Delta[y_1/x_1, \ldots, y_p/x_p]).\end{aligned}$$

We must now show that the removal of l does not affect the potential of the rest of the context, $\Gamma()$. By Lemma 6.4 we can remove any variables not in $\mathrm{FV}(e_i)$ from $\Gamma()$ and still get a typing for e_i. Thus the benign sharing condition (from page 9)

$$l \notin \mathcal{R}(\sigma, S[x_1 \mapsto v_1, \ldots, x_p \mapsto v_p] \upharpoonright \mathrm{FV}(e_i)) \tag{2.1}$$

shows that the removal of l does not affect the potential of the (remaining) context. We can then use Lemma 6.1 and the induction hypothesis as usual.

D-MATCH and D-CASE with E-MATCHN′ and E-MATCH′. As for the previous two cases, because the only difference is that l is not deallocated. So we need only note that l is not removed from σ, and thus the benign sharing argument is not required. □

As usual, we can use the theorem to show that the stack usage of the whole program respects the bounds obtained from a typing:

Corollary 6.8. *Let* $\text{size}(c) = 0$ *for all constructors c. Suppose a well typed program has an initial function* f*, and the arguments for* f *are given as values* $v_1, \ldots, v_p$ *with an initial store* σ*. If*

$$\Sigma(f) = \Gamma \to T', k_1' \mid \Phi'$$

where $\text{names}(\Gamma) = (y_1, \ldots, y_p)$ *then any execution of* $f(v_1, \ldots, v_p)$ *will require at most*

$$\Upsilon_{\mathsf{c}}(\sigma, [y_1 \mapsto v_1, \ldots, y_p \mapsto v_p], \Gamma) + \text{stack}(f)$$

units of memory, for any assignment of nonnegative rationals to constraint variables which satisfies Φ'*.*

Proof. The program is well typed, so from the function signature we have

$$\Gamma \vdash^{f,\mathsf{true}}_{\Sigma,F} e_f : T', k_1' \mid \Phi'.$$

Applying D-FUN we have

$$\Gamma, \text{stack}(f) \vdash^{\mathsf{initial},\mathsf{false}}_{\Sigma,F} f(y_1, \ldots, y_p) : T', k'' \mid \Phi' \cup \{k_1' + \text{stack}(f) \geq k''\},$$

and so by Theorem 6.7 no evaluation of $f(x_1, \ldots, x_p)$ requires more than

$$\Upsilon_{\mathsf{c}}(\sigma, [y_1 \mapsto v_1, \mapsto y_p \mapsto v_p], \Gamma) + \text{stack}(f)$$

units of stack space. □

As in the previous chapter, we can extend the soundness theorem to cover non-terminating programs using the partial evaluation extension to the operational semantics:

Theorem 6.9. *Let* $\text{size}(c) = 0$ *for all constructors c. If an expression e in some well-typed program has a typing*

$$\Gamma \vdash^{g,t}_{\Sigma,F} e : T, k' \mid \Phi$$

with an assignment of nonnegative rationals to constraint variables which satisfies the constraints in g's signature, and an evaluation in the extended operational semantics of Section 2.1.3

$$S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma'$$

satisfying the benign sharing conditions, and $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ *is defined, then for any* $q \in \mathbb{Q}^+$ *and* $m \in \mathbb{N}$ *such that*

$$m \geq \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma) + q$$

m will be a sufficient amount of stack space for the execution to succeed,

$$m, S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma', m,$$

where either $v = \mathsf{halted}$*, or*

$$m \geq \Upsilon_{\mathsf{t}}(\sigma', v, T) + k' + q.$$

Proof. By the proof of Theorem 6.7 combined with the extra cases for the partial evaluation semantics from the proof of Theorem 5.12 (which is the corresponding result for the previous system, on page 87). □

As a result, Corollary 6.8 for entire programs also holds for the partial evaluation semantics. Thus regardless of how long a non-terminating program is allowed to run for, it will never violate an inferred stack space bound.

6.4 Forming a linear program

We have shown that solving the constraint set generated by a typing of a program will yield an upper bound on the stack memory usage, but the constraints generated are not all linear. The D-SPLIT rule and the plus-contraction form of D-$\equiv$ both take fractions of the annotations in a context, producing quadratic constraints of the form $qk = k'$.

For the type annotations we will fix q as part of the type inference process, so that constraints such as $qk = k'$ are linear with q as a fixed coefficient. Our inference process is detailed in the next chapter, but we note two useful conventions that we will use there and in the examples. When the subcontexts involved are used in similar ways, it makes sense to split the potential equally, and so we choose q appropriately. For example, if we had a context $x : T,\ y : T$ and wanted an additive context using D-SPLIT we would take $q = \frac{1}{2}$ to get $x : \frac{1}{2}T,\ y : \frac{1}{2}T$.

The other common case is when we want all of the potential from one of the subcontexts and none from the rest, which can be realised with D-SPLIT using $q = 1$. We will see examples below where we use this to separate the structure of a tree from its contents.

Dealing with annotations for fixed amounts of potential which appear in the context is easier. We can use the following derived rule for plus-contraction:

Lemma 6.10. *The rule*

$$\frac{\Gamma(k_1,k_2) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi}{\Gamma(k) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \cup \{k = k_1 + k_2\}} \quad \textit{(D-CONTRACTA)}$$

is derivable modulo equivalence of constraints.

Proof. We can derive the following:

$$\frac{\Gamma(k_1,k_2) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \qquad \dfrac{\dfrac{}{qk = k_1 \,|\, \{qk = k_1\}} \qquad \dfrac{}{(1-q)k = k_2 \,|\, \{(1-q)k = k_2\}}}{k \cong k_1, k_2 \,|\, \{0 \le q, q \le 1, qk = k_1, (1-q)k = k_2\}}}{\Gamma(k) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \cup \{0 \le q, q \le 1, qk = k_1, (1-q)k = k_2\}} \text{D-}\equiv$$

It remains to show that the constraints above are satisfied by any solution to $k = k_1 + k_2$. For $k = 0$, we have $k_1 = k_2 = 0$ because all annotations are nonnegative by definition. Otherwise, take $q = k_1/k$, and the required constraint set becomes

$$\Phi \cup \{0 \le k_1, k_1 \le k, k_1 = k_1, k - k_1 = k_2\}.$$

This is equivalent to

$$\Phi \cup \{k = k_1 + k_2\},$$

as required. □

We also have the reverse rule:

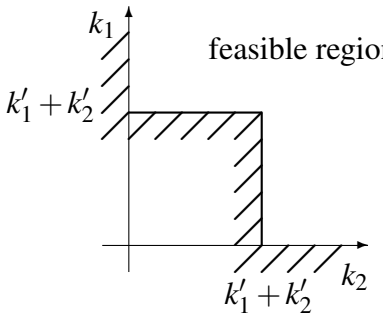
Lemma 6.11. *The rule*

$$\frac{\Gamma(k) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi}{\Gamma(k_1,k_2) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \cup \{k = k_1 + k_2\}} \quad \textit{(D-CONTRACTA}')$$

is derivable modulo equivalence of constraints.

Proof. As for Lemma 6.10, except that we use the symmetry form of D-$\equiv$ to swap k and k_1, k_2. □

We might hope for an analogous rule with linear constraints for D-SPLIT to transform a context $\Gamma(k_1;k_2)$ into one of the form $\Gamma(k_1,k_2)$, but the corresponding constraint is $\max\{k_1,k_2\} \ge k_1' + k_2'$. This is not convex, as we can see from the following graph:



Thus such constraints cannot be solved by linear programming.

If we fix the coefficients q for type annotations and use the derived rule for annotations for fixed potential then our constraint set will be linear and can be solved by normal linear programming techniques as before.

6.5 Examples

To make our examples more readable we use the derived rules above, plus the following rule:

Lemma 6.12. *The rule*

$$\frac{\Gamma((\Delta_1;\ldots;\Delta_n),k) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi}{\Gamma((\Delta_1,k_1);\ldots;(\Delta_n,k_n)) \vdash^{g,t}_{\Sigma,F} e : T, k' \,|\, \Phi \cup \{k_i \geq k : \forall i \leq n\}}\ (\text{D-FACTORA})$$

is derivable, up to equivalent constraints.

Proof. For each Δ_i, k_i subcontext we can use the D-CONTRACTA rule to obtain Δ_i, k_i', k with the constraint $k_i = k_i' + k$, and then use D-WEAKEN to remove the k_i'. The k_i' annotation does not appear anywhere else, so the constraint can be replaced with $k_i \geq k$. Finally, using D-$\equiv$ with the distribution equivalence, $\Gamma, (\Delta; \Delta') \cong (\Gamma, \Delta); (\Gamma, \Delta')$, the whole subcontext becomes $(\Delta_1, \ldots, \Delta_n), k$. □

In the example derivations we will also combine several uses of D-WEAKEN, and combine weakening with the leaf rules where it is obvious which sections of the context should be removed.

Our first example is a simple recursive function on boolean trees to demonstrate how we can obtain a stack space bound in terms of the input's depth.

Example 6.13. Consider the following function which computes the pointwise 'and' of two binary trees with boolean values at the nodes:

```
let andtrees(t1,t2) =
  match t1 with leaf -> leaf | node(l1,r1,v1) ->
  match t2 with leaf -> leaf | node(l2,r2,v2) ->
    let l = andtrees(l1,l2) in
    let r = andtrees(r1,r2) in
    let v = if v1 then v2 else false in
      node(l,r,v)
```

Evaluating `andtrees(t1,t2)` according to the operational semantics requires stack space proportional to the length of longest path from the root which is common to both trees. The definition of the potential functions determines the form of the bounds the analysis can produce. Hence, the bounds for `andtrees` will be either the sum of values proportional to each trees depth, or the maximum of values proportional to each tree's depth. Thus some reasonable bounds we expect to be able to obtain using the new type system are stack(`andtrees`) times one of $|\texttt{t1}|_{\mathrm{d}}$, $|\texttt{t2}|_{\mathrm{d}}$ or $\max\{|\texttt{t1}|_{\mathrm{d}}, |\texttt{t2}|_{\mathrm{d}}\}$, where stack$(f)$ is our usual notation for the stack frame size of a function f. The max bound is potentially worse than the other two, but is provided to illustrate how a single function can have several signatures with different structures and different typing derivations.

First, let us consider obtaining a bound with the signature

$$\Sigma(\texttt{andtrees}) = \texttt{t1} : \mathsf{booltree}(k_1), \texttt{t2} : \mathsf{booltree}(k_2) \to \mathsf{booltree}(k_3), k'.$$

A typing for the function body using this signature is presented in Figure 6.6, where $T_i = \mathsf{booltree}(k_i)$. We have used some structural rules between the D-MATCH and D-LET rules to group the corresponding parts of the two trees together in the context, along with the fixed potential. Thus at each use of D-LET we can confine our interest to the subcontext containing precisely what we need to construct the new variable. Finally, to construct the new tree node we use two derived structural rules (D-FACTORA and D-CONTRACTA) to move the fixed potential to the required parts of the context.

$$\mathcal{D}_{\mathsf{leaf}_1} = \dfrac{\dfrac{\dfrac{}{\mathtt{t2}:T_2, 0 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{leaf} : T_3, k' \mid \Phi_1}\,\text{D-Construct}}{\mathtt{t2}:T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{leaf} : T_3, k' \mid \Phi_1}\,\text{D-}\equiv}{(), \mathtt{t2}:T_2 \mid T_1 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{leaf} \to \cdots : T_3, k' \mid \Phi_1}\,\text{D-Case}$$

$$\mathcal{D}_{\mathsf{leaf}_2} = \dfrac{\dfrac{}{(\mathtt{l1}:T_1; \mathtt{r1}:T_1; \mathtt{v1}:\mathsf{bool}), k_1 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{leaf} : T_3, k' \mid \Phi_2}\,\text{D-Construct}}{(\mathtt{l1}:T_1; \mathtt{r1}:T_1; \mathtt{v1}:\mathsf{bool}), k_1, () \mid T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{leaf} \to \cdots : T_3, k' \mid \Phi_2}\,\text{D-Case}$$

$$\mathcal{D}_{\mathsf{and}} = \dfrac{\dfrac{}{\mathtt{v2}:\mathsf{bool}, k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \mathtt{v2}:\mathsf{bool}, k_4 \mid \Phi_3}\,\text{D-Var} \qquad \dfrac{}{\mathtt{v2}:\mathsf{bool}, k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \mathtt{false}:\mathsf{bool}, k_4 \mid \Phi_3}\,\text{D-Bool}}{\mathtt{v1}:\mathsf{bool}, \mathtt{v2}:\mathsf{bool}, k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \mathtt{if\ v1} \cdots : \mathsf{bool}, k_4 \mid \Phi_3}\,\text{D-If}$$

$$\mathcal{D} = \dfrac{\mathcal{D}_{\mathsf{and}} \quad \dfrac{\dfrac{\dfrac{}{(\mathtt{l}:T_3; \mathtt{r}:T_3; \mathtt{v}:\mathsf{bool}), k_3, k_7 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{node} \cdots : T_3, k' \mid \Phi_4}\,\text{D-Construct}}{(\mathtt{l}:T_3; \mathtt{r}:T_3; \mathtt{v}:\mathsf{bool}), k_6 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{node} \cdots : T_3, k' \mid \Phi_{4,5}}\,\text{D-ContractA}}{(\mathtt{l}:T_3, k_5); (\mathtt{r}:T_3, k_5); (\mathtt{v}:\mathsf{bool}, k_4) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{node} \cdots : T_3, k' \mid \Phi_{4,5,6}}\,\text{D-FactorA}}{(\mathtt{l}:T_3, k_5); (\mathtt{r}:T_3, k_5); (\mathtt{v1}:\mathsf{bool}, \mathtt{v2}:\mathsf{bool}, k) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{let\ v} \cdots : T_3, k' \mid \Phi_{3,\ldots,6}}\,\text{D-Let}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{l1}:T_1, \mathtt{l2}:T_2, k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \mathtt{andtrees(l1,l2)} : T_3, k_5 \mid \Phi_7}\,\text{D-FunDef} \quad \dfrac{\dfrac{}{\mathtt{r1}:T_1, \mathtt{r2}:T_2, k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \mathtt{andtrees(r1,r2)} : T_3, k_5 \mid \Phi_7}\,\text{D-FunDef} \quad \mathcal{D}}{(\mathtt{l}:T_3, k_5); (\mathtt{r1}:T_1, \mathtt{r2}:T_2, k); (\mathtt{v1}:\mathsf{bool}, \mathtt{v2}:\mathsf{bool}, k) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{let\ r} \cdots : T_3, k' \mid \Phi_{3,\ldots,7}}\,\text{D-Let}}{(\mathtt{l1}:T_1, \mathtt{l2}:T_2, k); (\mathtt{r1}:T_1, \mathtt{r2}:T_2, k); (\mathtt{v1}:\mathsf{bool}, \mathtt{v2}:\mathsf{bool}, k) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{let\ l} \cdots : T_3, k' \mid \Phi_{3,\ldots,7}}\,\text{D-Let}}{(\mathtt{l1}:T_1, (\mathtt{l2}:T_2; \mathtt{r2}:T_2; \mathtt{v2}:\mathsf{bool}), k); (\mathtt{r1}:T_1, (\mathtt{l2}:T_2; \mathtt{r2}:T_2; \mathtt{v2}:\mathsf{bool}), k); (\mathtt{v1}:\mathsf{bool}, (\mathtt{l2}:T_2; \mathtt{r2}:T_2; \mathtt{v2}:\mathsf{bool}), k) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{let\ l} \cdots : T_3, k' \mid \Phi_{3,\ldots,7}}\,\text{D-Weaken}}{(\mathtt{l1}:T_1; \mathtt{r1}:T_1; \mathtt{v1}:\mathsf{bool}), (\mathtt{l2}:T_2; \mathtt{r2}:T_2; \mathtt{v2}:\mathsf{bool}), k \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{let\ l} \cdots : T_3, k' \mid \Phi_{3,\ldots,7}}\,\text{D-}\equiv}{(\mathtt{l1}:T_1; \mathtt{r1}:T_1; \mathtt{v1}:\mathsf{bool}), k_1, (\mathtt{l2}:T_2; \mathtt{r2}:T_2; \mathtt{v2}:\mathsf{bool}), k_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{let\ l} \cdots : T_3, k' \mid \Phi_{3,\ldots,8}}\,\text{D-ContractA}'}{(\mathtt{l1}:T_1; \mathtt{r1}:T_1; \mathtt{v1}:\mathsf{bool}), k_1, () \mid T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{node} \cdots : T_3, k' \mid \Phi_{3,\ldots,8}}\,\text{D-Case}$$

$$\dfrac{\mathcal{D}_{\mathsf{leaf}_2} \qquad \dfrac{(\mathtt{l1}:T_1; \mathtt{r1}:T_1; \mathtt{v1}:\mathsf{bool}), k_1, t2:T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{match\ t2} \cdots : T_3, k' \mid \Phi_{2,\ldots,8}}{(), \mathtt{t2}:T_2 \mid T_1 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{node} \cdots : T_3, k' \mid \Phi_{2,\ldots,8}}\,\text{D-Case}}{\mathtt{t1}:T_1, \mathtt{t2}:T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \mathtt{match\ t1} \cdots : T_3, k' \mid \Phi_{1,\ldots,8}}\,\text{D-Match}}{\mathcal{D}_{\mathsf{leaf}_1}}\,\text{D-Match}$$

Figure 6.6: A 'depth' typing for the body of `andtrees`

The constraints generated by this typing are

$\Phi_1 = \{0 \geq k'\}$ from D-CONSTRUCT,
$\Phi_2 = \{k_1 \geq k'\}$ from D-CONSTRUCT,
$\Phi_3 = \{k \geq k_4\}$ from D-VAR and D-BOOL,
$\Phi_4 = \{k_7 \geq k'\}$ from D-CONSTRUCT,
$\Phi_5 = \{k_6 = k_3 + k_7\}$ from D-CONTRACTA,
$\Phi_6 = \{k_4 \geq k_6,\ k_5 \geq k_6\}$ from D-FACTORA,
$\Phi_7 = \{k \geq \text{stack}'(\texttt{andtrees}, \texttt{andtrees}, \mathsf{false}),\ k + k' \geq k_5\}$ from D-FUNDEF,
$\Phi_8 = \{k = k_1 + k_2\}$ from D-CONTRACTA$'$.

and the following solution gives us the $\text{stack}(\texttt{andtrees}) \times |\texttt{t1}|_\text{d}$ bound,

$$k_1 = k = k_4 = k_5 = k_6 = k_3 = \text{stack}(\texttt{andtrees}), k_2 = k_7 = k' = 0,$$

with the signature

$$\texttt{t1} : \mathsf{booltree}(\text{stack}(\texttt{andtrees})), \texttt{t2} : \mathsf{booltree}(0) \to \mathsf{booltree}(\text{stack}(\texttt{andtrees})), 0.$$

Another solution gives us the $\text{stack}(\texttt{andtrees}) \times |\texttt{t2}|_\text{d}$ bound,

$$k_2 = k = k_4 = k_5 = k_6 = k_3 = \text{stack}(\texttt{andtrees}), k_1 = k_7 = k' = 0,$$

with the signature

$$\texttt{t1} : \mathsf{booltree}(0), \texttt{t2} : \mathsf{booltree}(\text{stack}(\texttt{andtrees})) \to \mathsf{booltree}(\text{stack}(\texttt{andtrees})), 0.$$

The bound expressed as a maximum requires a different signature,

$$\Sigma(\texttt{andtrees}) = \texttt{t1} : \mathsf{booltree}(k_1); \texttt{t2} : \mathsf{booltree}(k_2) \to \mathsf{booltree}(k_3), k'.$$

and a different typing (Figure 6.7). The constraints generated by this typing are

$\Phi_1 = \{0 \geq k'\}$ from D-CONSTRUCT,
$\Phi_2 = \{k_1 \geq k'\}$ from D-CONSTRUCT,
$\Phi_3 = \{k \geq k_4\}$ from D-VAR and D-BOOL,
$\Phi_4 = \{k_7 \geq k'\}$ from D-CONSTRUCT,
$\Phi_5 = \{k_6 = k_3 + k_7\}$ from D-CONTRACTA,
$\Phi_6 = \{k_4 \geq k_6,\ k_5 \geq k_6\}$ from D-FACTORA,
$\Phi_7 = \{k \geq \text{stack}'(\texttt{andtrees}, \texttt{andtrees}, \mathsf{false}),\ k + k' \geq k_5\}$ from D-FUNDEF,
$\Phi_8 = \{k_1 \geq k,\ k_2 \geq k\}$ from D-FACTORA.

$$\mathcal{D}_{\mathsf{leaf}_1} = \dfrac{\dfrac{\dfrac{}{\texttt{t2} : T_2, 0 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{leaf} : T_3, k' \mid \Phi_1}\,\text{D-Construct}}{\texttt{t2} : T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{leaf} : T_3, k' \mid \Phi_1}\,\text{D-}\equiv}{();\texttt{t2} : T_2 \mid T_1 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{leaf} \to \cdots : T_3, k' \mid \Phi_1}\,\text{D-Case}$$

$$\mathcal{D}_{\mathsf{leaf}_2} = \dfrac{\dfrac{}{(\texttt{l1} : T_1; \texttt{r1} : T_1; \texttt{v1} : \mathsf{bool}), k_1 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{leaf} : T_3, k' \mid \Phi_2}\,\text{D-Construct}}{((\texttt{l1} : T_1; \texttt{r1} : T_1; \texttt{v1} : \mathsf{bool}), k_1); () \mid T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{leaf} \to \cdots : T_3, k' \mid \Phi_2}\,\text{D-Case}$$

$$\mathcal{D}_{\mathsf{and}} = \dfrac{\dfrac{}{\texttt{v2} : \mathsf{bool}, k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \texttt{v2} : \mathsf{bool}, k_4 \mid \Phi_3}\,\text{D-Var} \qquad \dfrac{}{\texttt{v2} : \mathsf{bool}, k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \texttt{false} : \mathsf{bool}, k_4 \mid \Phi_3}\,\text{D-Bool}}{(\texttt{v1} : \mathsf{bool}; \texttt{v2} : \mathsf{bool}), k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \texttt{if v1} \cdots : \mathsf{bool}, k_4 \mid \Phi_3}\,\text{D-If}$$

$$\mathcal{D} = \dfrac{(\texttt{l} : T_3, k_5); (\texttt{r} : T_3, k_5); ((\texttt{v1} : \mathsf{bool}; \texttt{v2} : \mathsf{bool}), k) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{let v} \cdots : T_3, k' \mid \Phi_{3,\ldots,6}}{(\texttt{l} : T_3, k_5); (\texttt{r} : T_3, k_5); ((\texttt{v1} : \mathsf{bool}; \texttt{v2} : \mathsf{bool}), k) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{let v} \cdots : T_3, k' \mid \Phi_{3,\ldots,6}}\,\text{D-Let}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\texttt{l1} : T_1; \texttt{l2} : T_2), k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \texttt{andtrees(l1,l2)} : T_3, k_5 \mid \Phi_7}\,\text{D-FunDef} \quad \dfrac{\dfrac{}{(\texttt{r1} : T_1; \texttt{r2} : T_2), k \vdash^{\mathsf{false}}_{\Sigma,\emptyset} \texttt{andtrees(r1,r2)} : T_3, k_5 \mid \Phi_7}\,\text{D-FunDef} \quad \mathcal{D}}{(\texttt{l} : T_3, k_5); ((\texttt{r1} : T_1; \texttt{r2} : T_2), k); ((\texttt{v1} : \mathsf{bool}; \texttt{v2} : \mathsf{bool}), k) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{let l} \cdots : T_3, k' \mid \Phi_{3,\ldots,8}}\,\text{D-Let}}{((\texttt{l1} : T_1; \texttt{l2} : T_2), k); ((\texttt{r1} : T_1; \texttt{r2} : T_2), k); ((\texttt{v1} : \mathsf{bool}; \texttt{v2} : \mathsf{bool}), k) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{let l} \cdots : T_3, k' \mid \Phi_{3,\ldots,8}}\,\text{D-Let}}{(\texttt{l1} : T_1, k_1); (\texttt{r1} : T_1, k_1); (\texttt{v1} : \mathsf{bool}, k_1); (\texttt{l2} : T_2, k_2); (\texttt{r2} : T_2, k_2); (\texttt{v2} : \mathsf{bool}, k_2) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{let l} \cdots : T_3, k' \mid \Phi_{3,\ldots,8}}\,\text{D-FactorA}}{((\texttt{l1} : T_1; \texttt{r1} : T_1; \texttt{v1} : \mathsf{bool}), k_1); ((\texttt{l2} : T_2; \texttt{r2} : T_2; \texttt{v2} : \mathsf{bool}), k_2) \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{let l} \cdots : T_3, k' \mid \Phi_{3,\ldots,8}}\,\text{D-}\equiv}{((\texttt{l1} : T_1; \texttt{r1} : T_1; \texttt{v1} : \mathsf{bool}), k_1); () \mid T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{node} \cdots : T_3, k' \mid \Phi_{3,\ldots,8}}\,\text{D-Case}}{((\texttt{l1} : T_1; \texttt{r1} : T_1; \texttt{v1} : \mathsf{bool}), k_1); \texttt{t2} : T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{match t2} \cdots : T_3, k' \mid \Phi_{2,\ldots,8}}\,\text{D-Case}}{();\texttt{t2} : T_2 \mid T_1 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{node} \cdots : T_3, k' \mid \Phi_{2,\ldots,8}}\,\text{D-Match}}{\texttt{t1} : T_1; \texttt{t2} : T_2 \vdash^{\mathsf{true}}_{\Sigma,\emptyset} \texttt{match t1} \cdots : T_3, k' \mid \Phi_{1,\ldots,8}}\,\text{D-Match}$$

Figure 6.7: A max-'depth' typing for the body of `andtrees`

Only Φ_8 has changed from the previous typing, but we now have the solution

$$k = k_1 = k_2 = k_3 = k_4 = k_5 = k_6 = \text{stack}(\texttt{andtrees}),\ k_7 = k' = 0,$$

and the signature

$$\texttt{t1} : \text{booltree}(\text{stack}(\texttt{andtrees})); \texttt{t2} : \text{booltree}(\text{stack}(\texttt{andtrees})) \to \text{booltree}(\text{stack}(\texttt{andtrees})), 0,$$

representing the bound

$$\max\{\text{stack}(\texttt{andtrees}) \times |\texttt{t1}|_{\text{d}},\ \text{stack}(\texttt{andtrees}) \times |\texttt{t2}|_{\text{d}}\}.$$

Our next example demonstrates how our new 'maximum' form of bounds can avoid splitting potential between different appearances of a variable unnecessarily.

Example 6.14. Consider the following function

```
let maybetail(l,b) = match l with cons(h,t)'-> if b then t else l
```

which returns either the tail of the list, or the whole list.

It requires only a constant amount of stack space, which all of our analyses easily handle. However, consider composing it with a function such as `notlist`, which uses a linear amount of stack space:

```
let maybenot(l,b) = let l' = maybetail(l,b) in notlist l'
```

The analysis of `maybetail` is responsible for turning the bound with respect to $|\texttt{l}'|$ into a bound with respect to $|\texttt{l}|$. The `maybetail` function's signature has the form

$$\texttt{maybetail} : \text{list}(k_1), \text{bool}, n_1 \to \text{list}(k_2), n_2,$$

The analysis will produce constraints forcing $k_2 \times |\texttt{l}'|$ to provide the linear part of the bound for `notlist`, $\text{stack}(\texttt{notlist}) \times |\texttt{l}'|$. It also produces constraints when analysing `maybetail` to ensure that k_1 is large enough so that

$$k_1 \times |\texttt{l}| \geq k_2 \times |\texttt{l}'|.$$

As we know that `l` and `l'` differ by at most one element, we also know that $k_1 = k_2 = \text{stack}(\texttt{notlist})$ is sufficient. We will see that the constraints produced by our new type system allow this tight bound, but those produced by the previous analyses can only give a larger bound, $k_1 = 2 \times k_2 = 2 \times \text{stack}(\texttt{notlist})$.

In the type systems for our previous analyses we must use the contraction rule SHARE before the first use of `l` for MATCH, dividing the potential between it (including `t`) and the second use of `l`. See Figure 6.8 for the typing in the direct adaption of the Hofmann-Jost system of Chapter 4 (the give-back system is similar). However, only one of `t` and `l` is used, so the remainder of the potential is ignored by the rest of the analysis. Thus in a larger program where some memory requirement proportional to the result of `maybetail` appears, such as `maybenot`, the bound expressed in terms of the argument to `maybetail` will be overestimated by a factor of two because for any k the best bound from Hofmann-Jost is

$$\texttt{maybetail} : \mathsf{list}(2 \times k), \mathsf{bool}, 0 \to \mathsf{list}(k), 0.$$

However, in the depth type system we can use max-contraction to avoid splitting up the potential. A typing for the body of `maybetail` in the depth system is presented in Figure 6.9. Note that the potential of the result can now be as large as the potential of the argument:

$$\texttt{maybetail} : \texttt{l} : \mathsf{list}(k), \texttt{b} : \mathsf{bool} \to \mathsf{list}(k), 0,$$

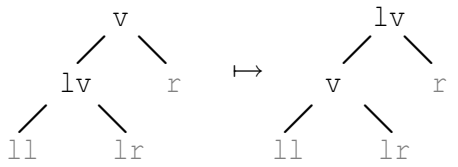
for any k. Thus the analysis of `maybetail` faithfully translates parts of the bound expressed in terms of its result's depth into a bound with respect to its argument's depth.

Finally, we consider an example which illustrates a limitation of the depth type system.

Example 6.15. Consider the following function which swaps the root value with its left child's value in a tree:

```
let swapleft t = match t with node(l,r,v) ->
                   match l with node(ll,lr,lv) ->
                   let l' = node (ll,lr,v) in
                     node(l',r,lv)
```

That is, it performs this transformation:



$$\mathcal{D}_t = \dfrac{\dfrac{}{\mathtt{h} : \mathsf{bool}, \mathtt{t} : \mathsf{list}(k'), \mathtt{l}_2 : \mathsf{list}(k_2), \mathtt{b} : \mathsf{bool}, n_1 \vdash_{\Sigma,F} \mathtt{t} : \mathsf{list}(k'), n' \mid \Phi_1}\,\textsc{Var}}{\mathtt{h} : \mathsf{bool}, \mathtt{t} : \mathsf{list}(k_1), \mathtt{l}_2 : \mathsf{list}(k_2), \mathtt{b} : \mathsf{bool}, n_1 \vdash_{\Sigma,F} \mathtt{t} : \mathsf{list}(k'), n' \mid \Phi_{1,2}}\,\textsc{Share}$$

$$\mathcal{D}_l = \dfrac{\dfrac{}{\mathtt{h} : \mathsf{bool}, \mathtt{t} : \mathsf{list}(k_1), \mathtt{l}_2 : \mathsf{list}(k'), \mathtt{b} : \mathsf{bool}, n_1 \vdash_{\Sigma,F} \mathtt{l}_2 : \mathsf{list}(k'), n' \mid \Phi_1}\,\textsc{Var}}{\mathtt{h} : \mathsf{bool}, \mathtt{t} : \mathsf{list}(k_1), \mathtt{l}_2 : \mathsf{list}(k_2), \mathtt{b} : \mathsf{bool}, n_1 \vdash_{\Sigma,F} \mathtt{l}_2 : \mathsf{list}(k'), n' \mid \Phi_{1,3}}\,\textsc{Share}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}_t \qquad \mathcal{D}_l}{\mathtt{h} : \mathsf{bool}, \mathtt{t} : \mathsf{list}(k_1), \mathtt{l}_2 : \mathsf{list}(k_2), \mathtt{b} : \mathsf{bool}, n_1 \vdash_{\Sigma,F} \mathtt{if} \cdots : \mathsf{list}(k'), n' \mid \Phi_{1,\ldots,3}}\,\textsc{If}}{\mathtt{l}_2 : \mathsf{list}(k_2), \mathtt{b} : \mathsf{bool}, n \triangleright \mathsf{list}(k_1) \vdash_{\Sigma,F} \mathtt{cons} \cdots : \mathsf{list}(k'), n' \mid \Phi_{1,\ldots,4}}\,\textsc{Case}}{\mathtt{l}_1 : \mathsf{list}(k_1), \mathtt{l}_2 : \mathsf{list}(k_2), \mathtt{b} : \mathsf{bool}, n \vdash_{\Sigma,F} \mathtt{match} \cdots : \mathsf{list}(k'), n' \mid \Phi_{1,\ldots,4}}\,\textsc{Match}}{\mathtt{l} : \mathsf{list}(k), \mathtt{b} : \mathsf{bool}, n \vdash_{\Sigma,F} \mathtt{match} \cdots : \mathsf{list}(k'), n' \mid \Phi_{1,\ldots,5}}\,\textsc{Share}$$

$$\Phi_1 = \{n_1 \geq n'\},\ \Phi_2 = \{k_1 \geq k'\},\ \Phi_3 = \{k_2 \geq k'\},\ \Phi_4 = \{n_1 = n + k_1\},\ \Phi_5 = \{k = k_1 + k_2\}$$

Figure 6.8: A plain Hofmann-Jost typing for `maybetail`

$$\mathcal{D}_t = \dfrac{\dfrac{\dfrac{}{\mathtt{t} : \mathsf{list}(k'), n \vdash_{\Sigma,F} \mathtt{t} : \mathsf{list}(k'), n' \mid \Phi_1}\,\text{D-}\textsc{Var}}{\mathtt{t} : \mathsf{list}(k), n \vdash_{\Sigma,F} \mathtt{t} : \mathsf{list}(k'), n' \mid \Phi_{1,2}}\,\text{D-}\textsc{WeakenA}}{(((\mathtt{h} : \mathsf{bool}; \mathtt{t} : \mathsf{list}(k_1)), k); \mathtt{l} : \mathsf{list}(k)), n \vdash_{\Sigma,F} \mathtt{t} : \mathsf{list}(k'), n' \mid \Phi_{1,2}}\,\text{D-}\textsc{Weaken}$$

$$\mathcal{D}_l = \dfrac{\dfrac{\dfrac{}{\mathtt{l} : \mathsf{list}(k'), n \vdash_{\Sigma,F} \mathtt{l} : \mathsf{list}(k'), n' \mid \Phi_1}\,\text{D-}\textsc{Var}}{\mathtt{l} : \mathsf{list}(k), n \vdash_{\Sigma,F} \mathtt{l} : \mathsf{list}(k'), n' \mid \Phi_{1,2}}\,\text{D-}\textsc{WeakenA}}{(((\mathtt{h} : \mathsf{bool}; \mathtt{t} : \mathsf{list}(k_1)), k); \mathtt{l} : \mathsf{list}(k)), n \vdash_{\Sigma,F} \mathtt{l} : \mathsf{list}(k'), n' \mid \Phi_{1,2}}\,\text{D-}\textsc{Weaken}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}_t \qquad \mathcal{D}_l}{(((\mathtt{h} : \mathsf{bool}; \mathtt{t} : \mathsf{list}(k)), k); \mathtt{l} : \mathsf{list}(k)), \mathtt{b} : \mathsf{bool}, n \vdash_{\Sigma,F} \mathtt{if} \cdots : \mathsf{list}(k'), n' \mid \Phi_{1,2}}\,\text{D-}\textsc{If}}{((); \mathtt{l} : \mathsf{list}(k)), \mathtt{b} : \mathsf{bool}, n \triangleright \mathsf{list}(k) \vdash_{\Sigma,F} \mathtt{cons} \cdots : \mathsf{list}(k'), n' \mid \Phi_{1,2}}\,\text{D-}\textsc{Case}}{(\mathtt{l} : \mathsf{list}(k); \mathtt{l} : \mathsf{list}(k)), \mathtt{b} : \mathsf{bool}, n \vdash_{\Sigma,F} \mathtt{match} \cdots : \mathsf{list}(k'), n' \mid \Phi_{1,2}}\,\text{D-}\textsc{Match}}{\mathtt{l} : \mathsf{list}(k), \mathtt{b} : \mathsf{bool}, n \vdash_{\Sigma,F} \mathtt{match} \cdots : \mathsf{list}(k'), n' \mid \Phi_{1,2}}\,\text{D-}\equiv$$

$$\Phi_1 = \{n \geq n'\},\ \Phi_2 = \{k \geq k'\}$$

Figure 6.9: A ‘depth’ typing for `maybetail`

Again, this function obviously runs in constant stack space (as any of the stack analyses we have examined can determine), and so we are interested in how the analysis of `swapleft` affects the bounds of a larger program. It does not change the depth of the tree, or its contents, but only the positions of the contents. Thus if `swapleft` t produces the tree t', and we have a bound $k \times |t'|_{\mathrm{d}}$ on some later part of the program, does the analysis of `swapleft` give us the tight bound of $k \times |t|_{\mathrm{d}}$?

If the values at the nodes are booleans then we can, and such a typing is presented in Figure 6.10. However, it uses a trick: the values at the nodes have no potential because they are booleans, and so we can use D-SPLIT to change the context

$$(\mathtt{l} : \mathsf{tree}(k); \mathtt{r} : \mathsf{tree}(k); \mathtt{v} : \mathsf{bool}), k$$

into

$$(\mathtt{l} : \mathsf{tree}(k); \mathtt{r} : \mathsf{tree}(k)), \mathtt{v} : \mathsf{bool}, k$$

with $q = 1$ for the `l;r` subcontext, and $1 - q = 0$ for `v`. Note that both contexts have the same potential, and so no inflation of the overall bounds occurs.

If we consider richer tree types, such as the tree of lists introduced in Section 6.1, then the situation is more complex. When the tree of lists was introduced we explained that the form of the bounds was not merely an annotation times the depth of the tree, but the maximum cost of following a path through the tree and one of the lists, *weighted* by the annotations. If we know in advance that the lists' annotation will be zero then the form of the bounds degenerates to the depth of the tree, and we can use the D-SPLIT trick again.

In general, however, we want to be able to express bounds which include the lists, and thus must be able to assign potential to them. The best we can do at present is to overestimate the cost of `swapleft` by k units of potential, inflating the overall bound by at least k units for every use of `swapleft`. This is necessary and sufficient because if v has enough potential then it will determine the potential of the entire tree. Thus swapping v with lv will make the overall potential v's potential *plus* k because v is one step away from the root of the tree.

Ideally we would like to separate such data structures into layers by defining the potential function so as to sum the maximum potential from the structural 'layer' and the maximum potential of the content 'layer'. We leave this to future work (see Section 9.2.1).

$$\mathcal{D}_{\mathtt{l}'} = \cfrac{\cfrac{}{(\mathtt{ll} : \mathsf{tree}(k_2); \mathtt{lr} : \mathsf{tree}(k_2); \mathtt{v} : \mathsf{bool}), k, 0 \vdash_{\Sigma,F} \mathtt{node} \cdots : \mathsf{tree}(k_2), 0 \mid \Phi_1}\;\textsc{D-Construct}}{(\mathtt{ll} : \mathsf{tree}(k_2); \mathtt{lr} : \mathsf{tree}(k_2); \mathtt{v} : \mathsf{bool}), k \vdash_{\Sigma,F} \mathtt{node} \cdots : \mathsf{tree}(k_2), 0 \mid \Phi_1}\;\textsc{D-}\equiv$$

$$\cfrac{\mathcal{D}_{\mathtt{l}'} \qquad \cfrac{\cfrac{}{(\mathtt{l}' : \mathsf{tree}(k_2); \mathtt{r} : \mathsf{tree}(k_2); \mathtt{lv} : \mathsf{bool}), k_2, 0 \vdash_{\Sigma,F} \mathtt{node} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_2}\;\textsc{D-Construct}}{((\mathtt{l}' : \mathsf{tree}(k_2), 0); \mathtt{r} : \mathsf{tree}(k_2); \mathtt{lv} : \mathsf{bool}), k_2 \vdash_{\Sigma,F} \mathtt{node} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_2}\;\textsc{D-}\equiv}{(((\mathtt{ll} : \mathsf{tree}(k_2); \mathtt{lr} : \mathsf{tree}(k_2); \mathtt{v} : \mathsf{bool}), k_2); \mathtt{r} : \mathsf{tree}(k_2); \mathtt{lv} : \mathsf{bool}), k_2 \vdash_{\Sigma,F} \mathtt{let} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2}}\;\textsc{D-Let}$$

$$\cfrac{\cdots}{((\mathtt{ll} : \mathsf{tree}(k_2), k_2); (\mathtt{lr} : \mathsf{tree}(k_2), k_2); (\mathtt{v} : \mathsf{bool}, k_2); \mathtt{r} : \mathsf{tree}(k_2); \mathtt{lv} : \mathsf{bool}), k_2 \vdash_{\Sigma,F} \mathtt{let} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2}}\;\textsc{D-}\equiv$$

$$\cfrac{\cdots}{\begin{pmatrix} (\mathtt{ll} : \mathsf{tree}(k_2), k_2, \mathtt{v} : \mathsf{bool}); (\mathtt{lr} : \mathsf{tree}(k_2), k_2, \mathtt{v} : \mathsf{bool}); (\mathtt{lv} : \mathsf{bool}, k_2, \mathtt{v} : \mathsf{bool}); \\ (\mathtt{lv} : \mathsf{bool}, k_2, \mathtt{v} : \mathsf{bool}); (\mathtt{r} : \mathsf{tree}(k_2), \mathtt{v} : \mathsf{bool}) \end{pmatrix}, k_2 \vdash_{\Sigma,F} \mathtt{let} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2}}\;\textsc{D-Weaken}$$

$$\cfrac{\cdots}{(((\mathtt{ll} : \mathsf{tree}(k_2); \mathtt{lr} : \mathsf{tree}(k_2); \mathtt{lv} : \mathsf{bool}), k_2); \mathtt{r} : \mathsf{tree}(k_2)), \mathtt{v} : \mathsf{bool}, k_2 \vdash_{\Sigma,F} \mathtt{let} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2}}\;\textsc{D-}\equiv$$

$$\cfrac{\cdots}{(((\mathtt{ll} : \mathsf{tree}(k_3); \mathtt{lr} : \mathsf{tree}(k_3); \mathtt{lv} : \mathsf{bool}), k_3); \mathtt{r} : \mathsf{tree}(k_3)), \mathtt{v} : \mathsf{bool}, k_1 \vdash_{\Sigma,F} \mathtt{let} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2,3}}\;\textsc{D-WeakenA}$$

$$\cfrac{\cdots}{(((\mathtt{ll} : \mathsf{tree}(k_1); \mathtt{lr} : \mathsf{tree}(k_1); \mathtt{lv} : \mathsf{bool}), k_1); \mathtt{r} : \mathsf{tree}(k_1); \mathtt{v} : \mathsf{bool}), k_1 \vdash_{\Sigma,F} \mathtt{let} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2,3,4}}\;\textsc{D-Split}$$

$$\cfrac{\cdots}{((); \mathtt{r} : \mathsf{tree}(k_1); \mathtt{v} : \mathsf{bool}), k_1 \shortmid \mathsf{tree}(k_1) \vdash_{\Sigma,F} \mathtt{node} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2,3,4}}\;\textsc{D-Case}$$

$$\cfrac{\cdots}{(\mathtt{l} : \mathsf{tree}(k_1); \mathtt{r} : \mathsf{tree}(k_1); \mathtt{v} : \mathsf{bool}), k_1 \vdash_{\Sigma,F} \mathtt{match} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2,3,4}}\;\textsc{D-Match}$$

$$\cfrac{\cfrac{\cdots}{() \shortmid \mathsf{tree}(k_1) \vdash_{\Sigma,F} \mathtt{node} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2,3,4}}\;\textsc{D-Case}}{\mathtt{t} : \mathsf{tree}(k_1) \vdash_{\Sigma,F} \mathtt{match} \cdots : \mathsf{tree}(k_2), k' \mid \Phi_{1,2,3,4}}\;\textsc{D-Match}$$

$$\Phi_1 = \{0 \geq 0\},\ \Phi_2 = \{0 \geq k'\},\ \Phi_3 = \{k_3 \geq k_2,\ k_1 \geq k_2\},\ \Phi_4 = \{1 \times k_1 = k_3\}.$$

Figure 6.10: A depth typing for the body of `swapleft` on boolean trees

Chapter 7

Structural inference for the depth analysis

The previous chapter presented most of our new stack space analysis that expresses bounds using addition and maxima, and thus allows bounds on stack space in terms of the depth of data structures. We have a type system which can produce a set of arithmetic constraints as before, and solutions for the constraint system provide upper bounds on the stack space usage. We have also provided some techniques to ensure that we can construct typings where the constraint set forms a linear program, and so can be solved with standard linear programming techniques, as with our previous analyses.

In this chapter we investigate the remaining problem: how to obtain a typing by choosing where to use the non-syntax-directed structural rules. With such a typing we can proceed as above to obtain a stack memory bound. Note that the typings we generate cannot be the 'most general' in any rigorous sense — we have already seen in Example 6.13 that a single function can have two quite different typing derivations. Therefore we will aim to construct an inference procedure which performs well in general, without performing an expensive search for the absolutely optimal typing.

To make the problem more tractable and eliminate some of the choices like those in Example 6.13 we assume that the structure of the function signatures is supplied by the user of the analysis. We will discuss possible approaches to inferring the signatures briefly in Section 9.2.4. We also assume that the constructor signatures are given by the user, although it is easy to produce these automatically by defining them to yield potential proportional to the depth. More precisely, for a constructor c of unannotated

signature $T_1, \ldots, T_p \to ty$ we would define

$$\Sigma(c) = \forall \overline{k}.(x_1 : \widehat{T_1}; \ldots; x_p : \widehat{T_p}), k \to ty(\overline{k}),$$

where $\widehat{T_i}$ is T_i plus any annotations required, and $\overline{k}$ contains all of the annotations that appear in a constructor for $ty(\overline{k})$ including those in each $\widehat{T_i}$.

We will define our inference algorithm as a translation from expressions in LFD to expressions in LFD enriched by terms which describe the inferred changes in the typing context's structure, and by implication the typing rules required. The algorithm is based on generating a 'desired' context for each expression, starting with the innermost subexpressions. The major difficulty is to decide on the changes to the context structure (and thus the terms to be added) to accommodate binding expressions, and to bridge the gap between the 'desired' context for the function body and the given function signature.

7.1 The enriched language

To make the typing entirely deterministic we add new terms as described above, and add a little information to the existing binding terms. Thus an enriched expression specifies the precise form of the typing derivation for the original expression in the type system defined in Chapter 6.

In that chapter we took the two context formers, ',' and ';', to be implicitly associative. Here we wish to remove all doubt as to which subcontexts we are manipulating and so we treat the formers as constructing lists of subcontexts (*bunches*), which may be nested. We denote an additive bunch as $(\Gamma_1, \ldots, \Gamma_n)$, a maximal bunch as $\{\Gamma_1; \ldots; \Gamma_n\}$, and interpret an empty list, $()$ or $\{\}$, as an empty context. This notation is slightly different from the previous chapter, but makes the context manipulation involved clearer. Thus two of our new terms will deal with grouping and ungrouping contexts, even though associativity is implicit in the type system.

We also need to give the position of a subcontext, for which we use a sequence of positions in successive nested contexts. For example, the position of y in

$$\{\underbrace{(x : T_1, k_1, \overbrace{\{z : T_2; y : T_3\}}^{3})}_{1}; k_2\}$$

is $(1,3,2)$. To give the position of the hole in a context-with-a-hole $\Gamma()$ we define a

function π:

$$\begin{aligned}\pi(()) &= (),\\ \pi((\Gamma_1,\ldots,\Gamma_i(),\ldots,\Gamma_n)) &= (i)\cdot\pi(\Gamma_i()),\\ \pi(\{\Gamma_1;\ldots;\Gamma_i();\ldots;\Gamma_n\}) &= (i)\cdot\pi(\Gamma_i()).\end{aligned}$$

We can now define our new terms, accompanied by a justification that the term represents a derivation in the type system. For each term we give one or more pseudo-typing rules similar to those in the previous chapter, omitting the Σ, F, g, t from $\vdash^{g,t}_{\Sigma,F}$ and the constraint set because they have no relevance to the soundness of the rules for the new terms. The main purpose of our new rules is to present the precise context transformation for each term.

The new terms and their justifications are:

$$\frac{\Gamma((\Gamma_1,\ldots,(\Gamma_i,\ldots,\Gamma_{i+m-1}),\ldots,\Gamma_n)) \vdash e : T,k}{\Gamma((\Gamma_1,\ldots,\Gamma_i,\ldots,\Gamma_{i+m-1},\ldots,\Gamma_n)) \vdash \mathsf{group}(\pi(\Gamma()),i,m)\ \mathsf{in}\ e : T,k}$$

$$\frac{\Gamma(\{\Gamma_1;\ldots;\{\Gamma_i;\ldots;\Gamma_{i+m-1}\};\ldots;\Gamma_n\}) \vdash e : T,k}{\Gamma(\{\Gamma_1;\ldots;\Gamma_i;\ldots;\Gamma_{i+m-1};\ldots;\Gamma_n\}) \vdash \mathsf{group}(\pi(\Gamma()),i,m)\ \mathsf{in}\ e : T,k}$$

$$\frac{\Gamma((\Gamma_1,\ldots,\Gamma_i,\ldots,\Gamma_{i+m-1},\ldots,\Gamma_n)) \vdash e : T,k}{\Gamma((\Gamma_1,\ldots,(\Gamma_i,\ldots,\Gamma_{i+m-1}),\ldots,\Gamma_n)) \vdash \mathsf{ungroup}(\pi(\Gamma()),i)\ \mathsf{in}\ e : T,k}$$

$$\frac{\Gamma(\{\Gamma_1;\ldots;\Gamma_i;\ldots;\Gamma_{i+m-1};\ldots;\Gamma_n\}) \vdash e : T,k}{\Gamma(\{\Gamma_1;\ldots;\{\Gamma_i;\ldots;\Gamma_{i+m-1}\};\ldots;\Gamma_n\}) \vdash \mathsf{ungroup}(\pi(\Gamma()),i)\ \mathsf{in}\ e : T,k}$$

The group and ungroup terms are justified by the implicit associativity of the context formers in the depth system.

$$\frac{\Gamma((\Gamma_{\rho(1)},\ldots,\Gamma_{\rho(n)})) \vdash e : T,k \qquad \rho \text{ is a permutation on } 1,\ldots,n}{\Gamma((\Gamma_1,\ldots,\Gamma_n)) \vdash \mathsf{rearrange}(\pi(\Gamma()),\rho)\ \mathsf{in}\ e : T,k}$$

$$\frac{\Gamma(\{\Gamma_{\rho(1)};\ldots;\Gamma_{\rho(n)}\}) \vdash e : T,k \qquad \rho \text{ is a permutation on } 1,\ldots,n}{\Gamma(\{\Gamma_1;\ldots;\Gamma_n\}) \vdash \mathsf{rearrange}(\pi(\Gamma()),\rho)\ \mathsf{in}\ e : T,k}$$

The rearrange term is justified by repeated application of the plus-commutate and max-commute cases of D-$\equiv$.

$$\frac{\Gamma(\{(\Gamma_1,\ldots,\Gamma_{i-1},\Delta_1,\Gamma_{i+1},\ldots,\Gamma_m);\ldots;(\Gamma_1,\ldots,\Gamma_{i-1},\Delta_n,\Gamma_{i+1},\ldots,\Gamma_m)\}) \vdash e : T,k}{\Gamma((\Gamma_1,\ldots,\Gamma_{i-1},\{\Delta_1;\ldots;\Delta_n\},\Gamma_{i+1},\ldots,\Gamma_m)) \vdash \mathsf{distribute}(\pi(\Gamma()),i)\ \mathsf{in}\ e}$$

$$\frac{\Gamma((\Gamma_1,\ldots,\Gamma_i,\{(\Delta_{1,1},\ldots,\Delta_{1,m_1});\ldots;(\Delta_{n,1},\ldots,\Delta_{n,m_n})\},\Gamma'_1,\ldots,\Gamma'_j)) \vdash e : T,k \quad \text{Each } \rho_i \text{ is a substitution on constraint variables}}{\Gamma\left(\left\{\begin{array}{c}\left(\rho_1(\Gamma_1),\ldots,\rho_1(\Gamma_i),\Delta_{1,1},\ldots,\Delta_{1,m_1},\rho_1(\Gamma'_1),\ldots,\rho_1(\Gamma'_j)\right);\\ \ldots;\\ \left(\rho_n(\Gamma_1),\ldots,\rho_n(\Gamma_i),\Delta_{n,1},\ldots,\Delta_{n,m_n},\rho_n(\Gamma'_1),\ldots,\rho_n(\Gamma'_j)\right)\end{array}\right\}\right) \vdash \mathsf{factor}(\pi(\Gamma()),i,j) \text{ in } e}$$

These terms combine the bidirectional distribution case of D-$\equiv$ with associativity and (in the case of factor) weakening of annotations using D-WEAKENA to make the common portions match.

$$\frac{\Gamma(\Delta) \vdash e : T,k}{\Gamma(\Gamma'(\Delta)) \vdash \mathsf{weaken}(\pi(\Gamma()),\pi(\Gamma'())) \text{ in } e : T,k}$$

$$\frac{\Gamma((\Delta_{j_1},\ldots,\Delta_{j_{m-n}})) \vdash e : T,k \quad \{j_1,\ldots,j_{m-n}\} = \{1,\ldots,m\} \setminus \{i_1,\ldots,i_n\}}{\Gamma((\Delta_1,\ldots,\Delta_m)) \vdash \mathsf{remove}(\pi(\Gamma()),\{i_1,\ldots,i_n\}) \text{ in } e : T,k}$$

$$\frac{\Gamma(\{\Delta_{j_1};\ldots;\Delta_{j_{m-n}}\}) \vdash e : T,k \quad \{j_1,\ldots,j_{m-n}\} = \{1,\ldots,m\} \setminus \{i_1,\ldots,i_n\}}{\Gamma(\{\Delta_1;\ldots;\Delta_m\}) \vdash \mathsf{remove}(\pi(\Gamma()),\{i_1,\ldots,i_n\}) \text{ in } e : T,k}$$

Both of these terms come from D-WEAKEN. For the remove term associativity allows us to group the subcontexts to be removed first.

$$\frac{\Gamma(\{\overbrace{\Delta;\ldots;\Delta}^{n \text{ times}}\}) \vdash e : T,k}{\Gamma(\Delta) \vdash \mathsf{maxcontract}(\pi(\Gamma()),n) \text{ in } e : T,k}$$

This is justified by repeated use of the max-contract case of D-$\equiv$.

$$\frac{\Gamma((q_1\Delta_1,\ldots,q_n\Delta_n)) \vdash e : T,k \qquad \sum_{i=1}^n q_i \leq 1}{\Gamma(\{\Delta_1;\ldots;\Delta_n\}) \vdash \mathsf{split}(\pi(\Gamma()),(q_1,\ldots,q_n)) \text{ in } e : T,k}$$

$$\frac{\Gamma(\Delta) \vdash e : T,k \qquad \sum_{i=1}^n q_i \leq 1}{\Gamma((q_1\Delta,\ldots,q_n\Delta)) \vdash \mathsf{unsplit}(\pi(\Gamma())) \text{ in } e : T,k}$$

The split term embodies the D-SPLIT rule. The scaling of contexts in the rule is informal — it is only the structure of the contexts and the limit on the sum of the q_i that is important to justify the term. The actual scaling is realised by the constraints generated

in the resulting typing. The unsplit case is justified by the (reverse) plus-contract case of D-$\equiv$.

$$\frac{\Gamma((\Delta_1,\ldots,k'_{j_1},\ldots)) \vdash e : T,k}{\Gamma((\Delta_1,\ldots,k_{i_1},\ldots)) \vdash \mathsf{mixfixed}(\pi(\Gamma()),\{i_1,\ldots,i_m\},\{j_1,\ldots,j_n\}) \text{ in } e : T,k}$$

This term combines the derived rules D-CONTRACTA and D-CONTRACTA$'$ from Section 6.4 to combine, split and rearrange the fixed amounts of potential in a additive bunch. The resulting typing will produce constraints equivalent to

$$k_{i_1} + \cdots + k_{i_m} = k'_{j_1} + \cdots + k'_{j_n}.$$

We can conclude from the new terms that

Lemma 7.1. *Given structured contexts* Γ *and* Δ *if we have*

$$\begin{array}{c} \Delta \vdash e_1 : T,k \\ \vdots \\ \Gamma \vdash e_2 : T,k \end{array}$$

where e_2 *is* e_1 *enriched by one or more of the terms above then there is a partial derivation*

$$\begin{array}{c} \Delta \vdash^{g,t}_{\Sigma,F} |e_1| : T,k \,|\, \Phi' \\ \vdots \\ \Gamma \vdash^{g,t}_{\Sigma,F} |e_2| : T,k \,|\, \Phi \end{array}$$

in the depth type system of Chapter 6, using the obvious erasure function $|\cdot|$ *to remove the new terms.*

Proof. For each of the new terms, we have given the corresponding typing rules for the corresponding partial derivation in the depth type system. □

Our final enrichment to the language is to add positions to the terms where we need to identify a particular subcontext. These are

$$\begin{aligned} e := {} & \ldots \\ & \mid \mathsf{let}\ x@P = e_1\ \mathsf{in}\ e_2 \\ & \mid \mathsf{match}\ x@P\ \mathsf{with}\ (x_1,x_2) \to e \\ & \mid \mathsf{match}\ x@P\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r \\ & \mid \mathsf{match}\ x@P\ \mathsf{with}\ p_1 \to e_1 \mid \cdots \mid p_m \to e_m \end{aligned}$$

where P is a subcontext position as defined above. For let it gives the position of the subcontext used to type e_1 and which is replaced by its result, x. For the other terms, it gives the position of the variable to match and replace.

7.2 An overview of the inference algorithm

At the beginning of this chapter we assumed that the user provides the context structure of the function signatures and the constructor signatures. This ensures that we know what we expect the structure of the context to look like for all of the innermost expressions (constants, variables, function applications, pairs, sums and datatype construction).

Thus the heart of the algorithm is to work outwards, deriving further 'desirable' typing contexts for compound expressions, until we have generated a desirable context for the entire function body. During this process we also add suitable terms to perform the required context transformations to recreate the desired context of each subexpression.

The most difficult part is to deal with binding. Given a desired context for a subexpression we must isolate the bound variables so that we obtain a context where they can be replaced according to the typing rule of the binding term. We also add terms to recreate the desired typing context for the subexpression. To help do this we will introduce a bidirectional context expansion transformation, which allows us to consider contexts in a maximum-of-sums form:

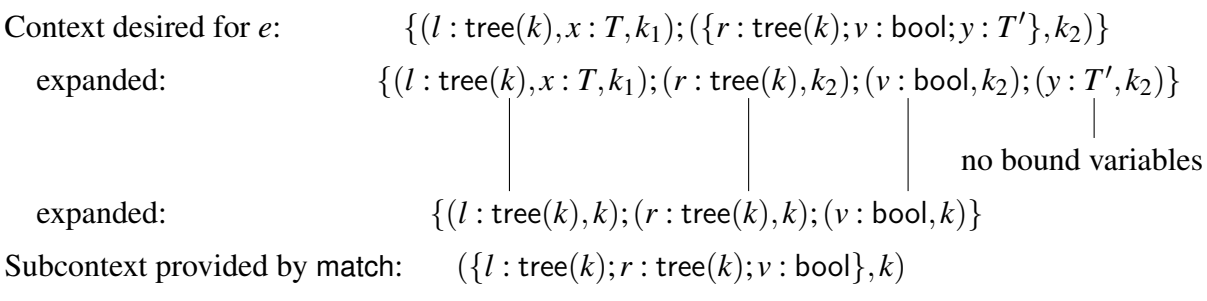
$$\{(x_{1,1} : T_{1,1}, \ldots); (x_{2,1} : T_{2,1}, k_{2,1}, \ldots); \ldots\},$$

and which adds terms to perform the expansion or contraction in the typing.

Thus when binding variables we only need to consider expanded contexts. For example, suppose we have the desired context

$$\{(l : \mathsf{tree}(k), x : T, k_1); (\{r : \mathsf{tree}(k); v : \mathsf{bool}; y : T'\}, k_2)\}$$

for the subexpression e in $\mathsf{match}\ t\ \mathsf{with}\ \mathsf{node}(l,r,v) \to e$. We can expand the context for e and the subcontext for the bound variables from the match, then decide how to match up the two:



This should be read in the same direction as a typing derivation — at the top is the context desired for typing e, at the bottom the subcontext introduced by the match containing the bound variables which will replace t. We are going to devise terms to add around e which will form the top context from the context provided by the typing of the match expression.

The above matching suggests that to construct the desired context for the match expression we should start with the expanded context for e, then remove the bound variables to construct a 'remainder' for each of these bunches consisting of the variables that were not bound and the fixed potential, add t to the context 'in addition' to all the remainders, and leave the bunches with no bound variables alone:

$$\{(t : \mathsf{tree}(k), \{(x : T, k_1'); (k_2')\}); (y : T', k_2)\}.$$

For the fixed amounts of potential we adopt the convention that they follow the variables in the same bunch in the expanded context. In this example, k_1 and two of the k_2 annotations are in bunches with bound variables. Thus we will add mixfixed terms to allow the potential to be supplied from either the k in the context from the match, or from their counterparts in the 'remainders', k_1' and k_2'. If a bunch has no variables at all, only fixed amounts of potential, then we attempt to find a fixed amount in the binding context to supply some or all of the potential, and have a remainder context with a fixed amount for the rest.

Finally, we need to match up the generated context for the entire function body with the function signature. Fortunately, this is similar to the handling of binding expressions, except that all variables are bound and there are no remainders.

Throughout the inference process every change in the desired contexts is mirrored by the addition of a context manipulation term. These represent the corresponding non-syntax-directed typing rules in the depth type system of Chapter 6 which perform the context changes. Erasing these extra terms from each function body yields the original expressions, and thus the inference process provides a typing of the original program.

7.3 Details of the inference algorithm

We now describe the structural inference algorithm in detail, beginning with the backbone which transforms each expression, then the method for dealing with binding, and

finally how to bridge the gap between the context generated by the inference and the function signature.

We assume that D-WEAKENA will be used to allow the weakening of each annotation before its use. This means that we do not have to worry about whether two annotations that must match in a typing should be the same constraint variable, and during the examples of the inference will simply give each new annotation a fresh constraint variable. We expect the structure of the signatures for constructors and functions to be supplied by the user. These take the same form as in Chapter 6, except that we do not yet know the constraint set for each function signature.

The transformation for expressions is given in Figures 7.1 and 7.2. It consists of a set of syntax-directed rules with judgements of the form

$$\Gamma \vdash_{\Sigma} e : T \mapsto \Delta, e',$$

where Γ is the plain unannotated typing context, e is an LFD expression, T is an unannotated type, Σ is the supplied, structured, signatures, Δ is the structured 'desired' context and e' is the expression in the enriched language. We use $\widehat{T}$ to denote the (unannotated) type T with fresh constraint variables inserted for the annotations. We describe the rules in order of increasing complexity.

The I-VAR case produces a context containing the variable, plus a fixed amount of potential k to accompany the result, and leaves the expression unchanged. This corresponds to the D-VAR typing rule. The function application case, I-FUN, is similar. It uses the context from the function signature with a suitable substitution of the variable names and also adds a fixed amount to pay for the stack space for the call and accompany the result type. The other innermost expressions (constants, pairs, sums and datatype construction) are treated in the same way.

Our first form of binding expression is let. The premises of I-LET use the auxiliary function isolate to deal with the binding. We will define this more precisely shortly, but informally

$$\text{isolate}(\Delta_1, \Delta_2, e) = (\Delta^{\text{m}}, \Delta^{+}, e')$$

changes Δ_2 to extract all of the variables bound in Δ_1, producing two subcontexts, Δ^{m} and Δ^{+}, such that $\{\Delta^{\text{m}}; (\Delta_1, \Delta^{+})\}$ will be transformed in the typing derivation into Δ_2 by the extra terms in e'. In I-LET we extract the result of e_1, $(x : T_0, k_0)$, and replace it by the context desired for e_1. We also add the position of that context to the enriched let expression. The pair-matching case is similar.

$$\frac{}{\Gamma \vdash_\Sigma * : 1 \mapsto k, *} \quad \text{(I-UNIT)}$$

$$\frac{c \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash_\Sigma c : \mathsf{bool} \mapsto k, c} \quad \text{(I-BOOL)}$$

$$\frac{}{\Gamma, x : T \vdash_\Sigma x : T \mapsto (x : \widehat{T}, k), x} \quad \text{(I-VAR)}$$

$$\frac{\Sigma(f) = \Delta \to \widehat{T}, k' \mid \Phi \qquad \mathrm{names}(\Delta) = (y_1, \ldots, y_p)}{\Gamma \vdash_\Sigma f(x_1, \ldots, x_p) : T \mapsto (\Delta[x_1/y_1, \ldots, x_p/y_p], k), f(x_1, \ldots, x_p)} \quad \text{(I-FUN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e_1 : T_0 \mapsto \Delta_1, e_1' \qquad \Gamma, x : T_0 \vdash_\Sigma e_2 : T \mapsto \Delta_2, e_2' \\ \mathrm{isolate}((x : \widehat{T_0}, k), \Delta_2, e_2') = (\Delta_2^{\mathrm{m}}, \Delta_2^{+}, e_2'')\end{array}}{\Gamma \vdash_\Sigma \mathsf{let}\ x : T_0 = e_1\ \mathsf{in}\ e_2 : T \mapsto \{\Delta_2^{\mathrm{m}}; (\Delta_1, \Delta_2^{+})\}, \mathsf{let}\ x@(2,1) = e_1'\ \mathsf{in}\ e_2''} \quad \text{(I-LET)}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : T \mapsto \Delta_1, e_1' \qquad \Gamma \vdash_\Sigma e_2 : T \mapsto \Delta_2, e_2'}{\begin{array}{l}\Gamma \vdash_\Sigma \mathsf{if}\ x\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2' : T \\ \quad \mapsto \{x : \mathsf{bool}; \Delta_1; \Delta_2\}, \mathsf{if}\ x\ \mathsf{then}\ \mathsf{weaken}((), (2))\ \mathsf{in}\ e_1'\ \mathsf{else}\ \mathsf{weaken}((), (3))\ \mathsf{in}\ e_2'\end{array}} \quad \text{(I-IF)}$$

$$\frac{}{\Gamma, x_1 : T_1, x_2 : T_2 \vdash_\Sigma (x_1, x_2) : T_1 \otimes T_2 \mapsto (x_1 : \widehat{T_1}, x_2 : \widehat{T_2}, k), (x_1, x_2)} \quad \text{(I-PAIR)}$$

$$\frac{\begin{array}{c}\Gamma(x) = T_1 \otimes T_2 \qquad \Gamma, x_1 : T_1, x_2 : T_2 \vdash_\Sigma e : T \mapsto \Delta, e' \\ \mathrm{isolate}((x_1 : \widehat{T_1}, x_2 : \widehat{T_2}), \Delta, e') = (\Delta^{\mathrm{m}}, \Delta^{+}, e'')\end{array}}{\begin{array}{l}\Gamma \vdash_\Sigma \mathsf{match}\ x@(2,1)\ \mathsf{with}\ (x_1, x_2) \to e : T \\ \quad \mapsto \{\Delta^{\mathrm{m}}; (x : \widehat{T_1} \otimes \widehat{T_2}, \Delta^{+})\}, \mathsf{match}\ x\ \mathsf{with}\ (x_1, x_2) \to e''\end{array}} \quad \text{(I-MATCHPAIR)}$$

$$\frac{}{\Gamma, x : T_1 \vdash_\Sigma \mathsf{inl}(x) : (T_1 + T_2) \mapsto (x : T_1, k_1, k), \mathsf{inl}(x)} \quad \text{(I-INL)}$$

$$\frac{}{\Gamma, x : T_2 \vdash_\Sigma \mathsf{inr}(x) : (T_1 + T_2) \mapsto (x : T_2, k_2, k), \mathsf{inr}(x)} \quad \text{(I-INR)}$$

$$\frac{\begin{array}{c}\Gamma(x) = T_1 + T_2 \\ \Gamma, x_1 : T_1 \vdash_\Sigma e_1 : T \mapsto \Delta_1, e_1' \qquad \mathrm{isolate}((x_1 : \widehat{T_1}, k_1), \Delta_1, e_1') = (\Delta_1^{\mathrm{m}}, \Delta_1^{+}, e_1'') \\ \Gamma, x_2 : T_2 \vdash_\Sigma e_2 : T \mapsto \Delta_2, e_2' \qquad \mathrm{isolate}((x_2 : \widehat{T_2}, k_2), \Delta_2, e_2') = (\Delta_2^{\mathrm{m}}, \Delta_2^{+}, e_2'')\end{array}}{\begin{array}{l}\Gamma \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_1) \to e_1 \mid \mathsf{inr}(x_2) \to e_2 : T \\ \quad \mapsto \{\Delta_1^{\mathrm{m}}; \Delta_2^{\mathrm{m}}; (x : \widehat{T_1 + T_2}, \{\Delta_1^{+}; \Delta_2^{+}\})\}, \\ \qquad \mathsf{match}\ x@(3,1)\ \mathsf{with}\ \mathsf{inl}(x_1) \to \mathsf{weaken}((3,2),(1))\ \mathsf{in}\ \mathsf{remove}((), \{2\})\ \mathsf{in}\ e_1'' \\ \qquad\qquad \mid \mathsf{inr}(x_2) \to \mathsf{weaken}((3,2),(2))\ \mathsf{in}\ \mathsf{remove}((), \{1\})\ \mathsf{in}\ e_2''\end{array}} \quad \text{(I-MATCHSUM)}$$

Figure 7.1: The skeleton of the structural inference algorithm

$$\frac{\Sigma(c) = \forall \bar{k}.\Delta \to ty(\bar{k}) \qquad \text{names}(\Delta) = (y_1, \ldots, y_p)}{\Gamma \vdash_\Sigma c(x_1, \ldots, x_p) : ty \mapsto (\Delta[x_1/y_1, \ldots, x_p/y_p], k), c(x_1, \ldots, x_p)} \text{(I-CONSTRUCT)}$$

$$\text{for all } i, 1 \leq i \leq n. \begin{cases} \Sigma(c_i) = \forall \bar{k}.\Delta_i \to ty(\bar{k}) \qquad \text{names}(\Delta_i) = (y_1, \ldots, y_{p_i}) \\ |\Delta_i| = T_{i,1}, \ldots, T_{i,p_i} \\ \Gamma, x_{i,1} : T_{i,1}, \ldots, x_{i,p_i} : T_{i,p_i} \vdash_\Sigma e_i : T \mapsto \Delta_i', e_i' \\ \text{isolate}(\Delta_i[x_{i,1}/y_{i,1}, \ldots, x_{i,p_i}/y_{i,p_i}], \Delta_i', e_i') = (\Delta_i^{\text{m}}, \Delta_i^+, e_i'') \end{cases}$$

$$\begin{aligned} &\Gamma \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ c_1(x_{1,1}, \ldots, x_{1,p_1})\langle' \rangle \to e_1 \mid \cdots \mid c_n(x_{n,1}, \ldots, x_{n,p_n})\langle' \rangle \to e_n : T \\ &\quad \mapsto \{\{\Delta_1^{\text{m}}; \ldots; \Delta_n^{\text{m}}\}; (x : ty(\bar{k}), \{\Delta_1^+, \ldots, \Delta_n^+\})\}, \\ &\qquad \mathsf{match}\ x@(2,1)\ \mathsf{with} \\ &\qquad\quad c_1(x_{1,1}, \ldots, x_{1,p_1})\langle' \rangle \to \mathsf{weaken}((2,2),(1))\ \mathsf{in}\ \mathsf{weaken}((1),(1))\ \mathsf{in}\ e_1'' \\ &\qquad \mid \ldots \\ &\qquad \mid c_n(x_{n,1}, \ldots, x_{n,p_n})\langle' \rangle \to \mathsf{weaken}((2,2),(n))\ \mathsf{in}\ \mathsf{weaken}((1),(n))\ \mathsf{in}\ e_n'' \end{aligned} \text{(I-MATCH)}$$

Figure 7.2: The skeleton of the structural inference algorithm (continued)

The first form of conditional expression is if. The I-IF rule recursively treats each branch, then combines the desired contexts into a maximum bunch alongside the conditional variable, x. This reflects the potential requirements: we need enough potential for the maximum of the required potential of each branch. We add weakening terms to the branches to remove the other branch's context.

The remaining cases are I-MATCHSUM and I-MATCH. The $|\Delta|$ in the I-MATCH rule is the typing context Δ with the tree structure and annotations erased. These rules combine binding and conditionals. Thus for each branch we recursively deal with the subexpression and isolate the bound subcontext. We then combine all of the branches' contexts in maximum bunches and give an enriched term which, for the typing of each branch, removes all of the other branches' contexts and then uses the terms added by isolate to obtain the desired context for typing the subexpression.

7.3.1 Binding

We now move on to the definition of isolate. First, we define the bidirectional context expansion transformation. Given a context we can either enrich an expression to expand that context to a max-of-sums form, or enrich an expression to transform the

expanded context into the original.

For the former direction we repeatedly apply the following two rules until neither is applicable. If there is a subcontext which is an additive bunch containing a max bunch we add a distribute term:

$$(\Gamma_1,\ldots,\{\Delta_1;\ldots;\Delta_m\},\ldots,\Gamma_n) \mapsto \{(\Gamma_1,\ldots,\Delta_1,\ldots,\Gamma_n);\ldots;(\Gamma_1,\ldots,\Delta_m,\ldots,\Gamma_n)\}.$$

Moreover, where there is a nested bunch of the same sort we add ungroup to flatten it:

$$(\Gamma_1,\ldots,(\Gamma_i,\ldots,\Gamma_j),\ldots,\Gamma_n) \mapsto (\Gamma_1,\ldots,\Gamma_i,\ldots,\Gamma_j,\ldots,\Gamma_n)$$
$$\{\Gamma_1;\ldots;\{\Gamma_i;\ldots;\Gamma_j\};\ldots;\Gamma_n\} \mapsto \{\Gamma_1;\ldots;\Gamma_i;\ldots;\Gamma_j;\ldots;\Gamma_n\}$$

The result of repeating these actions is that none of the additive bunches will contain a maximum bunch, and there are no nested bunches of the same sort. Thus we have a max-of-sums form.

In the opposite direction we still expand the context in the same way with two rules, but add the factor and group terms instead of distribute and ungroup so that the enriched expression performs the opposite translation in the typing, from the expanded context to the original.

Note that simplifying the form of the context by using expansion comes at some cost: the expansion may yield a context which is exponentially large in the number of context former alterations (that is, the depth of alternate add/max bunch nesting). However, contexts are usually small, so we leave the possibility of reducing the amount of context expansion required to further work.

We can now define $\text{isolate}(\Delta_1,\Delta_2,e) = (\Delta^{\text{m}},\Delta^{+},e')$. Recall that the aim of this function is to produce a Δ^{m} and Δ^{+} for use in an expression's desired context such that $\{\Delta^{\text{m}};(\Delta_1,\Delta^{+})\}$ can be transformed to Δ_2 for typing the subcontext. First we consider the expanded contexts for the binding context, Δ_1', and the desired context for the subexpression, Δ_2'. Thus Δ_2' is composed for a number of additive bunches in a max-bunch:

$$\Delta_2' = \{\Delta_{2,1};\ldots;\Delta_{2,n}\} \quad \text{where each } \Delta_{2,i} \text{ is of the form} \quad \Delta_{2,i} = (x_{i,1} : T_{i,1},\ldots,k_{i,j},\ldots).$$

We check each $\Delta_{2,i}$ to see if it has any of the bound variables from Δ_1 or if it consists solely of fixed amounts of potential and Δ_1 provides some fixed amount. If neither of these holds, we do not change it and add it to Δ^{m}. Otherwise we wish to separate the bound variables and produce a 'remainder' context to put in Δ^{+}.

We can form the remainder, $\Delta'_{2,i}$, as the variables and fixed potential from $\Delta_{2,i}$ except for the bound variables. Thus, together with the binding context, the remainder is sufficient to recreate $\Delta_{2,i}$. The terms that we add will combine the fixed potential in the remainder with any supplied by the relevant parts of the binding context. We construct Δ^+ as the max-bunch of all the remainders.

To add the terms which transform each remainder $\Delta'_{2,i}$ into $\Delta_{2,i}$ we need to decide which parts of the expanded binding context Δ'_1 are required. The expanded binding context is of the form $\Delta'_1 = \{\Delta_{1,1}; \ldots; \Delta_{1,m}\}$, where each $\Delta_{1,j}$ is an additive bunch. Several of these $\Delta_{1,j}$ may contain variables which the $\Delta_{2,i}$ under consideration requires. (If we only need a fixed amount of potential, then we pick one $\Delta_{1,j}$ which has a fixed amount.) Moreover, in some cases *several* $\Delta_{1,j}$ may contain a single variable due to the expansion. We thus choose a set $\{j_1, \ldots, j_p\}$ of $\Delta_{1,j}$ which covers all of the bound variables that appear in $\Delta_{2,i}$[1]. The $\Delta_{2,i}$ we wish to recreate is a additive bunch, so we must split the chosen $\{\Delta_{1,j_1}; \ldots; \Delta_{1,j_p}\}$. As we have no local information about how the potential will be used, we do not know the best way to split up the potential. We thus choose to approximate it by splitting it up into equal proportions, yielding $\left(\frac{1}{p}\Delta_{1,j_1}, \ldots, \frac{1}{p}\Delta_{1,j_p}\right)$.

Now that we know Δ^{m} and Δ^+ and which parts of Δ'_1 to use for each bunch, we can form the enriched expression e' which specifies all of the typing context manipulations required to form Δ_2. Recall that we start with $\{\Delta^{\mathrm{m}}; (\Delta_1, \Delta^+)\}$. We proceed as follows:

1. First we have the terms which expand Δ_1 using distribute and ungroup giving $\{\Delta^{\mathrm{m}}; (\Delta'_1, \Delta^+)\}$ with $\Delta'_1 = \{\Delta_{1,1}; \ldots; \Delta_{1,m}\}$;
2. now we distribute Δ'_1 over the remainders in Δ^+;
3. for each remainder $\Delta'_{2,i}$ we need to transform $(\Delta'_1, \Delta'_{2,i})$ into $\Delta_{2,i}$:
   (a) We remove the unnecessary $\Delta_{1,j}$ bunches (those not in $\{j_1, \ldots, j_p\}$);
   (b) we split the other $\Delta_{1,j}$ bunches evenly as above and ungroup to give a single flat additive bunch;
   (c) we remove the unnecessary bound variables and unsplit any unnecessary duplicates (so as to retain as much potential as possible);

[1] Finding a smallest set covering is NP-hard, but we can get a reasonable approximation by choosing the $\Delta_{1,j}$ bunches which provide the greatest number of bound variables first, see (Cormen et al., 1990, Section 37.3) for more details on set covering.

(d) we split any bound variables that need to be duplicated (again, distributing potential evenly, as we have no way to tell locally which instances would benefit from a greater share of the potential);

(e) we add mixfixed to add any fixed potential from the binding context to the potential from the remainder;

(f) we rearrange the bunch to form $\Delta_{2,i}$.

4. We then use rearrange to place the unchanged bunches from Δ^{m} into the correct positions to form Δ_2';

5. and finally we use the reverse context expansion procedure above to form Δ_2, using factor and ungroup.

7.3.2 Function signature matching

The work we need to do to bridge the gap between the desired context generated for the function body and the function signature is very similar to the process for isolate. The difference is that we do not produce a Δ^{m} or Δ^{+}. Instead, all the variables are bound, and we obtain all the fixed amounts of potential from the signature (or fix them to zero if there is no suitable source).

Now let Δ_1 be the context from the signature, and Δ_2 be the desired context for the function body. We enrich the expression to

1. expand the signature context Δ_1 to Δ_1' as before;

2. use maxcontract to make a copy of Δ_1' to form each additive bunch of Δ_2' from;

3. create each $\Delta_{2,i}$ as before (remember that we do not have remainder bunches $\Delta_{2,i}'$ because all of the variables are bound); if we need a fixed amount of potential and have no source, use mixfixed with an empty 'source' set to fix it to zero;

4. use the reverse context expansion to form Δ_2.

7.3.3 Completing the inference

We perform the above procedure for each function body in the program and then use the type system to extract a linear program.

Theorem 7.2. *Given an unannotated typed program and the structured signatures for the functions and constructors, the inference process provides a typing in the depth type system of Chapter 6.*

Proof. In each enriched function body the additional terms describe the context manipulations required to make the desired context for the original expression. By Lemma 7.1 there exists a partial typing on the original program corresponding to these terms.

We can then move down through the original expression constructing a typing, using the terms added in the enriched version to decide where to use the non-syntax-directed rules in the same way, again by Lemma 7.1.

Taking the resulting typings for each function body, we obtain a typing of the entire program. □

Note that the inference process produces typings without quadratic constraints, as per Section 6.4. Thus, we can use standard linear programming techniques to find a bound on the stack memory usage, with the usual caveat that the linear program may not have any solutions if the program uses super-linear stack memory, or the stack memory usage is too subtle for the analysis to track.

This inference process has been implemented in Standard ML, along with a type checker for the enriched language which generates the linear program and presents the solutions. Separating the structural inference from the checker provides more confidence in the resulting bounds — any flaws in the implementation of the inference do not affect the soundness of any inferred bound, only our ability to obtain a bound and the precision of any inferred bounds.

7.4 Examples

We now revisit two of the examples from the previous chapter.

Example 7.3. Recall the `andtrees` function from Example 6.13 on page 115:

```
let andtrees(t1,t2) =
  match t1 with leaf -> leaf | node(l1,r1,v1) ->
  match t2 with leaf -> leaf | node(l2,r2,v2) ->
    let l = andtrees(l1,l2) in
    let r = andtrees(r1,r2) in
    let v = if v1 then v2 else false in
      node(l,r,v)
```

We illustrate the inference using the signature

$$\Sigma(\texttt{andtrees}) = (\texttt{t1} : \mathsf{booltree}(k), \texttt{t2} : \mathsf{booltree}(k')) \to \mathsf{booltree}(k''), k'''.$$

For each expression we give the original expression itself e_i, the desired context generated for it Δ_i, and the enriched expression e_i', see Figure 7.3.

Five cases involve binding (e_5, e_7, e_9, e_{11} and e_{13}). The let expressions are straightforward. For instance, for e_5 (`let v =`) the context for the second subexpression (Δ_1) expands to

$$\{(\texttt{l} : T_1, k_1, k_2); (\texttt{r} : T_1, k_1, k_2); (\texttt{v} : \mathsf{bool}, k_1, k_2)\}$$

and then we isolate the binding context $(\texttt{v} : \mathsf{bool}, k_5)$, which only affects the third bunch,

$$\mathsf{isolate}((\texttt{v} : \mathsf{bool}, k_5), \Delta_1, e_1') = (\{(\texttt{l} : T_1, k_1, k_2); (\texttt{r} : T_1, k_1, k_2)\}, \{(k_6)\}, e_1'')$$
$$\text{where } e_1'' = \mathsf{distribute}(\pi(2), 2) \text{ in } \mathsf{mixfixed}(\pi(2,1), \{2,3\}, \{2,3\}) \text{ in}$$
$$\mathsf{ungroup}((), 2) \text{ in } \mathsf{ungroup}((), 1) \text{ in } e_1'$$

and replace it by the context from the first subexpression, Δ_4 to give Δ_5. Note the mixfixed term, for which the corresponding typing will produce a constraint equivalent to

$$k_5 + k_6 = k_1 + k_2.$$

The other two let expressions (e_7 and e_9 are similar).

Each of the match expressions (e_{11} and e_{13}) has two cases, although both leaf cases are trivial. We trace the inference for e_{11}; first we expand the context from the node case, Δ_9, to get

$$\{(\texttt{r1} : T_2, \texttt{r2} : T_3, k_7, k_{11}); (\texttt{v1} : \mathsf{bool}); (\texttt{v2} : \mathsf{bool}, k_9); (k_{10}); (\texttt{l1} : T_4, \texttt{l2} : T_5, k_{12}, k_{13})\}.$$

We also expand the context from the constructor signature (substituting the correct variable names)

$$\Sigma(\texttt{node}) = \forall k.(\{l : T_k; r : T_k; v : \mathsf{bool}\}, k) \to T_k$$

to get

$$\Delta_{\mathsf{node}} = \{(\texttt{l2} : T_k, k); (\texttt{r2} : T_k, k); (\texttt{v2} : \mathsf{bool}, k)\}.$$

We pick out which bunches go into Δ_9^{m} and Δ_9^{+} by comparison with the expanded binding context. Only one involves no bound variables and is not entirely composed of fixed amounts of potential,

$$\Delta_9^{\mathrm{m}} = \{(\texttt{v1} : \mathsf{bool})\},$$

i	e_i	Δ_i	e'_i
1	`node(l,r,v)`	$(\{\texttt{l} : T_1; \texttt{r} : T_1; \texttt{v} : \mathsf{bool}\}, k_1, k_2)$	e_1
2	`false`	k_3	e_2
3	`v2`	$(\texttt{v2} : \mathsf{bool}, k_4)$	e_3
4	`if v1 ...`	$\{\texttt{v1} : \mathsf{bool}; (\texttt{v2} : \mathsf{bool}, k_4); k_3\}$	if `v1` then weaken$((),(2))$ in e_3 else weaken$((),(3))$ in e_2
5	`let v=...`	$\{\{(\texttt{l} : T_1, k_1, k_2); (\texttt{r} : T_1, k_1, k_2)\}; (\Delta_4, \{(k_6)\})\}$	let $\texttt{v}@(2,1) = e'_4$ in e''_1
6	`andtrees(r1,r2)`	$(\texttt{r1} : T_2, \texttt{r2} : T_3, k_7)$	e_6
7	`let r=...`	$\{\{(\texttt{l} : T_1, k_1, k_2); (\texttt{v1} : \mathsf{bool}); (\texttt{v2} : \mathsf{bool}, k_9); (k_{10})\}; (\Delta_6, \{(k_{11})\})\}$	let $\texttt{r}@(2,1) = e'_6$ in e''_5
8	`andtrees(l1,l2)`	$(\texttt{l1} : T_4, \texttt{l2} : T_5, k_{12})$	e_8
9	`let l=...`	$\{\{(\texttt{r1} : T_2, \texttt{r2} : T_3, k_7, k_{11}); (\texttt{v1} : \mathsf{bool}); (\texttt{v2} : \mathsf{bool}, k_9); (k_{10})\}; (\Delta_8, \{(k_{13})\})\}$	let $\texttt{l}@(2,1) = e'_8$ in e''_7
10	`leaf`	k_{14}	e_{10}
11	`match t2 ...`	$\left\{ \begin{array}{c} \{\{\}; \{(\texttt{v1} : \mathsf{bool})\}\}; \\ (\texttt{t2} : T_6, \{\{(k_{20})\}; \{(\texttt{r1} : T_2, k_{16}); (k_{17}); (k_{18}); (\texttt{l1} : T_4, k_{19})\}\}) \end{array} \right\}$	e'_{11} (see text)
12	`leaf`	k_{24}	e_{12}
13	`match t1 ...`	$\left\{ \begin{array}{c} \{\{\}; \{(\texttt{t2} : T_3, k_{20}); (\texttt{t2} : T_3, k_{17}); (\texttt{t2} : T_3, k_{18})\}\}; \\ (\texttt{t1} : T_7, \{\{(k_{22})\}; \{(); (\texttt{t2} : T_6, k_{21}); (\texttt{t2} : T_6, k_{23})\}\}) \end{array} \right\}$	e'_{13}

where $\quad T_i = \mathsf{booltree}(k'_i)$.

Figure 7.3: Results of the inference for expressions in `andtrees`

and for the rest we remove the bound variables and give fresh constraint variables for the fixed amounts (because they may be combined with the k from the binding context):

$$\Delta_9^+ = \{(\texttt{r1} : T_2, k_{16}); (k_{17}); (k_{18}); (\texttt{l1} : T_2, k_{19})\}.$$

The final context uses these, and the corresponding trivial contexts for the leaf case:

$$\Delta_{11} = \left\{ \begin{array}{c} \{\{\}; \{(\texttt{v1} : \mathsf{bool})\}\}; \\ (\texttt{t2} : T_6, \{\{(k_{20})\}; \{(\texttt{r1} : T_2, k_{16}); (k_{17}); (k_{18}); (\texttt{l1} : T_2, k_{19})\}\}) \end{array} \right\}$$

The corresponding enriched expression first discards the unused cases, then recreates the desired contexts as described above:

$$\begin{array}{l}
e'_{11} = \ \mathsf{match}\ \texttt{t2@}(2,1)\ \mathsf{with} \\
\qquad \mathsf{leaf} \rightarrow \ \mathsf{weaken}((2,2),(1),\mathsf{weaken}((1),(1),\mathsf{distribute}((2),2, \\
\qquad\qquad \mathsf{ungroup}((2,1),1,\mathsf{ungroup}((),2,\mathsf{ungroup}((),1,\mathsf{leaf})))))) \\
\qquad \mid \ \mathsf{node}(\texttt{l2},\texttt{l2},\texttt{v2}) \rightarrow \\
\qquad\qquad \textit{[remove subcontexts for leaf case]} \\
\qquad\qquad \mathsf{weaken}((2,2),(2))\ \mathsf{in}\ \mathsf{weaken}((1),(2))\ \mathsf{in} \\
\qquad\qquad \textit{[expand binding context and distribute across } \Delta_9^+\textit{]} \\
\qquad\qquad \mathsf{distribute}((2,1),1)\ \mathsf{in}\ \mathsf{distribute}((2),2)\ \mathsf{in} \\
\qquad\qquad \textit{[first subcontext]} \\
\qquad\qquad \mathsf{remove}((2,1,1),\{1,3\})\ \mathsf{in}\ \mathsf{ungroup}((2,1),1)\ \mathsf{in} \\
\qquad\qquad \mathsf{mixfixed}((2,1),\{2,4\},\{3,4\})\ \mathsf{in}\ \mathsf{rearrange}((2,1),(1\ 2))\ \mathsf{in} \\
\qquad\qquad \textit{[second subcontext]} \\
\qquad\qquad \mathsf{remove}((2,2,1),\{1,2\})\ \mathsf{in}\ \mathsf{ungroup}((2,2),1)\ \mathsf{in} \\
\qquad\qquad \mathsf{mixfixed}((2,2),\{2,3\},\{2\})\ \mathsf{in} \\
\qquad\qquad \textit{[third subcontext]} \\
\qquad\qquad \mathsf{remove}((2,3,1),\{1,3\})\ \mathsf{in}\ \mathsf{ungroup}((2,3),1)\ \mathsf{in} \\
\qquad\qquad \mathsf{remove}((2,3),\{1\})\ \mathsf{in}\ \mathsf{mixfixed}((2,3),\{1,2\},\{1\})\ \mathsf{in} \\
\qquad\qquad \textit{[fourth subcontext]} \\
\qquad\qquad \mathsf{remove}((2,4,1),\{2,3\})\ \mathsf{in}\ \mathsf{ungroup}((2,4),1)\ \mathsf{in} \\
\qquad\qquad \mathsf{mixfixed}((2,4),\{2,4\},\{3,4\})\ \mathsf{in}\ \mathsf{rearrange}((2,4),(1\ 2))\ \mathsf{in} \\
\qquad\qquad \textit{[flatten and rearrange context to get } \Delta'_9\textit{]} \\
\qquad\qquad \mathsf{ungroup}((),2)\ \mathsf{in}\ \mathsf{ungroup}((),1)\ \mathsf{in}\ \mathsf{rearrange}((),(1\ 2))\ \mathsf{in}
\end{array}$$

$$\begin{array}{l}
\textit{[reverse expansion of } \Delta_9\textit{]} \\
\mathsf{group}((),1,4)\ \mathsf{in}\ \mathsf{group}((),2,1)\ \mathsf{in} \\
\mathsf{factor}((2),3,0)\ \mathsf{in}\ \mathsf{group}((2),1,3)\ \mathsf{in}\ e'_9
\end{array}$$

Note that no bunch in the expanded version of Δ_9 contains more than one bound variables, so we do not need to split any contexts.

The other match expression, e_{13}, is similar.

Finally, we need to add terms to construct Δ_{13} from the function signature. We expand Δ_{13}, generating terms to recreate it from its expansion:

$$\Delta'_{13} = \left\{ \begin{array}{c} (\mathtt{t2} : T_6, k_{20}); (\mathtt{t2} : T_6, k_{17}); (\mathtt{t2} : T_6, k_{18}); (\mathtt{t1} : T_7, k_{22}); \\ (\mathtt{t1} : T_7); (\mathtt{t1} : T_7, \mathtt{t2} : T_6, k_{21}); (\mathtt{t1} : T_7, \mathtt{t2} : T_6, k_{23}) \end{array} \right\}$$

$$\begin{array}{rl}
e''_{13} = & \mathsf{group}((),1,3)\ \mathsf{in}\ \mathsf{group}((1),1,0)\ \mathsf{in}\ \mathsf{group}((1),2,3)\ \mathsf{in} \\
& \mathsf{group}((),2,4)\ \mathsf{in}\ \mathsf{factor}((2),1,0)\ \mathsf{in} \\
& \mathsf{group}((2,2),1,1)\ \mathsf{in}\ \mathsf{group}((2,2),2,3)\ \mathsf{in}\ e'_{13}
\end{array}$$

The inference process then produces a term for constructing the expanded context from the signature by making a copy of the signature for each bunch, then removing the unnecessary parts and making the correct number of fixed amounts for each bunch:

$$\begin{array}{rl}
e'_{\mathtt{andtrees}} = & \mathsf{maxcontract}((),7)\ \mathsf{in} \\
& \mathsf{remove}((1),\{1\})\ \mathsf{in}\ \mathsf{mixfixed}((1),\{\},\{2,3\})\ \mathsf{in} \\
& \mathsf{remove}((2),\{1\})\ \mathsf{in}\ \mathsf{mixfixed}((2),\{\},\{2,3\})\ \mathsf{in} \\
& \mathsf{remove}((3),\{1\})\ \mathsf{in}\ \mathsf{mixfixed}((3),\{\},\{2,3\})\ \mathsf{in} \\
& \mathsf{remove}((4),\{2\})\ \mathsf{in}\ \mathsf{mixfixed}((4),\{\},\{2,3\})\ \mathsf{in} \\
& \mathsf{remove}((5),\{2\})\ \mathsf{in}\ \mathsf{mixfixed}((5),\{\},\{2\})\ \mathsf{in} \\
& \mathsf{mixfixed}((6),\{\},\{2,4\})\ \mathsf{in} \\
& \mathsf{mixfixed}((7),\{\},\{2,4\})\ \mathsf{in}\ e''_{13}
\end{array}$$

We can then use the typing our enriched function body gives to generate a linear program. The solutions of the linear program then give us the same signatures as our manually-produced derivation:

$$\mathtt{andtrees} : \mathtt{t1} : \mathsf{booltree}(\mathsf{stack}(\mathtt{andtrees})), \mathtt{t2} : \mathsf{booltree}(0) \to \mathsf{booltree}(\mathsf{stack}(\mathtt{andtrees})), 0,$$
$$\mathtt{andtrees} : \mathtt{t1} : \mathsf{booltree}(0), \mathtt{t2} : \mathsf{booltree}(\mathsf{stack}(\mathtt{andtrees})) \to \mathsf{booltree}(\mathsf{stack}(\mathtt{andtrees})), 0.$$

The inference can also infer the maximum signature form of `andtrees` shown in Example 6.13.

The `maybetail` example is simpler:

Example 7.4. Recall the `maybetail` function from Example 6.14:

```
let maybetail(l,b) = match l with cons(h,t)'-> if b then t else l
```

The inference takes the form

i	e_i	Δ_i	e'_i
1	`l`	$(\texttt{l} : \mathsf{list}(k_1), k_2)$	`l`
2	`t`	$(\texttt{t} : \mathsf{list}(k_3), k_4)$	`t`
3	if `b`...	$\{\texttt{b} : \mathsf{bool}; (\texttt{t} : \mathsf{list}(k_3), k_4); (\texttt{l} : \mathsf{list}(k_1), k_2)\}$	if `b` then $\mathsf{weaken}((), 2)$ in e'_2 else $\mathsf{weaken}((), 3)$ in e'_1
4	match `l`...	$\left\{ \begin{array}{c} \{\{\texttt{b} : \mathsf{bool}; (\texttt{l} : \mathsf{list}(k_1), k_2)\}\}; \\ (\texttt{l} : \mathsf{list}(k_5), \{\{(k_4)\}\}) \end{array} \right\}$	e'_4

where

$$\begin{aligned} e'_4 = \mathsf{match}\ \texttt{l@}(2,1)\ \mathsf{with}\ \mathsf{cons}(\texttt{h},\texttt{t}) \to\ & \mathsf{weaken}((2,2),(1))\ \mathsf{in}\ \mathsf{weaken}((1),(1))\ \mathsf{in} \\ & \mathsf{distribute}((2),2)\ \mathsf{in}\ \mathsf{remove}((2,1,1),\{1\})\ \mathsf{in} \\ & \mathsf{ungroup}((2,1),1)\ \mathsf{in}\ \mathsf{mixfixed}((2,1),\{2,3\},\{2\})\ \mathsf{in} \\ & \mathsf{ungroup}((1),2)\ \mathsf{in}\ \mathsf{rearrange}((),(2\ 3))\ \mathsf{in}\ e'_3 \end{aligned}$$

which removes the extra grouping, removes `h` (which is never used), adds the potential from the list element to k_4 to get k_2 and flattens and rearranges the context to form Δ_3.

To get the function signature

$$\Sigma(\texttt{maybetail}) = (\texttt{l} : \mathsf{list}(k), \texttt{b} : \mathsf{bool}, n) \to \mathsf{list}(k'), n'$$

into the form of Δ_4 we first expand Δ_4,

$$\{\texttt{b} : \mathsf{bool}; (\texttt{l} : \mathsf{list}(k_1), k_2); (\texttt{l} : \mathsf{list}(k_5), k_4)\},$$

then make three copies of the signature, remove the unnecessary parts to form each additive bunch above, and add the terms to reverse the expansion:

$$\begin{aligned} e'_{\texttt{maybetail}} = \ & \mathsf{maxcontract}((), 3)\ \mathsf{in} \\ & \mathsf{remove}((1),\{1,3\})\ \mathsf{in}\ \mathsf{remove}((2),\{1\})\ \mathsf{in}\ \mathsf{remove}((3),\{1\})\ \mathsf{in} \\ & \quad \textit{[reverse the expansion to form } \Delta_4\textit{]} \\ & \mathsf{group}((),\{1,2\},1)\ \mathsf{in}\ \mathsf{group}((),\{2\},2)\ \mathsf{in} \\ & \mathsf{factor}((2),1,0)\ \mathsf{in}\ \mathsf{group}((2),2,2)\ \mathsf{in}\ \mathsf{group}((),1,1)\ \mathsf{in}\ e'_4 \end{aligned}$$

Type checking and constraint solving then yields the family of signatures from the previous chapter,

$$\texttt{maybetail} : \texttt{l} : \mathsf{list}(k), \texttt{b} : \mathsf{bool} \to \mathsf{list}(k), 0,$$

for any $k \in \mathbb{Q}^+$.

7.5 Rearranging tree contents

In Example 6.15 we saw the `swapleft` function which swaps the value at the top node of a binary tree with the value at its left child. Our concern was not the stack space usage of the function itself, but the relationship between the potential assigned to the argument and the result. This affects the overall bound because the relationship is used to translate the part of the bound expressed in terms of the size of `swapleft`'s result into one using the size of `swapleft`'s argument.

Recall that in the general case we needed to add an extra fixed amount of potential for every use of `swapleft` (that is, increase the overall bound) due to the form of the potential functions. We left dealing with the general case to future work, but gave a solution using D-SPLIT when the contents of the tree is not assigned any potential. We now extend the inference to use this trick.

First, let us recall the `swapleft` function itself:

```
let swapleft t = match t with node(l,r,v) ->
                 match l with node(ll,lr,lv) ->
                 let l' = node (ll,lr,v) in
                   node(l',r,lv)
```

When we apply the existing inference to the function it expands the desired context for the subexpression at `match l` and we find the two bunches

$$(\texttt{v} : \mathsf{bool}, k_1) \qquad \text{and} \qquad (\texttt{lv} : \mathsf{bool}, k_2).$$

The fixed amount k_2 must be at least as large as the per-node amount of potential for the result, because it is placed at the top of the tree. Similarly, k_1 must be at least *twice* that, because it is in the second level of the tree. As they are swapped over from the original argument, we need to supply an extra fixed amount of potential for k_1 to be large enough.

The idea of the extended inference is to split variables such as `v` and `lv` away from the rest of the context so that swapping them does not affect the potential required.

We first note that if we have some expression e typable in a context

$$\Gamma((\Delta, x : T)),$$

for a type T which cannot be assigned any potential, such as bool, then we can enrich e with $\mathsf{split}(\pi(\Gamma()),(1,0))$ and type it in the context

$$\Gamma(\{\Delta; x : T\}).$$

Dually, for the context $\Gamma(\{\Delta; x : T\})$ we can add maxcontract and weaken to type the enriched expression with the context

$$\Gamma((\Delta, x : T)).$$

Repeating this transformation and adding group terms to remove the nesting we can push $x : T$ up to the top of the context. That is, given e typed with $\Gamma(x : T)$ we can enrich e to type it in the context $\{\Gamma(\cdot); x : T\}$.

We thus define the extended inference to be the process defined in Section 7.3 except that it applies the above transformation at each point in the process where a variable of a type that cannot be assigned potential may be introduced into the middle of a desired context. That is, after I-VAR (to yield $\{x : T; k\}$ instead of $(x : T, k)$), I-CONSTRUCT, I-PAIR and I-LET.

Now we can infer the desired signature for `swapleft`:

Example 7.5. The extended inference procedure now produces an expanded context containing the variables for the contents of the nodes on their own,

$$(\mathtt{v} : \mathsf{bool}) \qquad \text{and} \qquad (\mathtt{lv} : \mathsf{bool}),$$

with the fixed amounts only present in bunches containing the tree structure (the variables `l`,`r`,`ll`,`lr` and `l'`). Thus `v` and `lv` can be swapped without requiring extra potential, yielding the overall signature of

$$\mathtt{swapleft} : \mathsf{booltree}(k) \to \mathsf{booltree}(k), 0 \quad \text{for any } k \in \mathbb{Q}^+.$$

We can also continue our heap sort example:

Example 7.6. The core function in our functional heap sort (introduced on page 29 and detailed in Appendix A) is `siftdown`. It takes two trees and an integer value as arguments and returns a single tree containing all three, maintaining the heap property. Checking usage requirements by hand we can see that this function requires stack space in proportion to the maximum depth of its arguments, and we would expect the analysis to be able to show this. Moreover, for the purposes of the analysis of the whole program it must also preserve the potential assigned to the trees, by requiring a fixed amount of potential to account for the possibility of an extra layer in the result.

Thus we start with the signature

$$\Sigma(\texttt{siftdown}) = (\{\texttt{t1} : \mathsf{inttree}(k_1); \texttt{t2} : \mathsf{inttree}(k_2); \texttt{v} : \mathsf{int}\}, k_3) \to \mathsf{inttree}(k_4), k_5.$$

Any overestimate of the potential required for this function will inflate the overall bound considerably, or may even make the generated constraints unsolvable, due to the repeated use of this function in the heap sort. Indeed, the analysis gives

$$\texttt{siftdown} : (\{\texttt{t1} : \mathsf{inttree}(7); \texttt{t2} : \mathsf{inttree}(7); \texttt{v} : \mathsf{int}\}, 14) \to \mathsf{inttree}(7), 0,$$

requiring $\max\{7 \times |\texttt{t1}|_{\mathrm{d}}, 7 \times |\texttt{t2}|_{\mathrm{d}}\} + 14$ units of potential. This is too large to give a bound on the whole sorting algorithm — the resulting linear program is infeasible. By simplifying the function, we can trace the problem to the base cases:

```
let siftdown(t1, t2, w) =
  let l = Leaf in
  match t1 with
    Leaf -> Node(l,l,w)
  | Node(t11,t12,v) ->
      (match t2 with
        Leaf ->
          if v < w then let a = Node(l,l,v) in Node(a,l,w)
                   else let a = Node(l,l,w) in Node(a,l,v)
...
```

The latter two cases require the extra 14 units of potential to construct the two nodes. The matching of `t1` should supply 7 of those. However, these cases use `l` to supply the leaf value because the program must be in let-normal form (that is, we cannot put `Leaf` directly into the `Node` expressions). We construct `l` once at the top of the function to avoid code duplication. The inference assumes that the potential required to construct

the nodes will be available at the point at which `l` is bound, but this is too early to use the potential from `t1`.

If we move the binding of `l` inwards then the inference will use the potential from matching `t1`:

```
let siftdown(t1, t2, w) =
  match t1 with
    Leaf -> let l = Leaf in Node(l,l,w)
  | Node(t11,t12,v) ->
      let l = Leaf in
      (match t2 with
        Leaf ->
          if v < w then let a = Node(l,l,v) in Node(a,l,w)
                   else let a = Node(l,l,w) in Node(a,l,v)
...
```

Note that this is closer to what a compiler's translation to let-normal form would produce, and so would be unlikely to be a problem when using a real programming language.

Thus we obtain the signature

$$\texttt{siftdown} : (\{\texttt{t1} : \mathsf{inttree}(7); \texttt{t2} : \mathsf{inttree}(7); \texttt{v} : \mathsf{int}\}, 7) \to \mathsf{inttree}(7), 0,$$

which is sufficient to go on to analyse the rest of the sorting algorithm. See Appendix A for details.

Chapter 8

Related work

Perhaps unsurprisingly, work specifically focused on inferring stack space bounds is rare. Heap space and execution time make more appealing targets, and only recently has the use of mobile code from untrustworthy sources become common enough to inspire interest in controlling *all* abusable resources. Thus our study of related work considers analyses of other resources — mostly heap space and execution time — and also of related techniques used for other purposes, such as optimisation.

In Section 4.3 we saw that heap space usage patterns tend to differ from stack space usage, which motivated the development of our later analyses. Similarly, when examining an analysis of execution time we must be aware that it differs considerably in behaviour from heap and stack space usage: it is monotonic and often super-linear.

We start by considering work closely related to Hofmann and Jost's heap space analysis, and then move on to work from further afield.

8.1 Hofmann-Jost based work

The Hofmann-Jost heap space analysis (Hofmann and Jost, 2003) and Jost's implementation (Jost, 2004b) formed the basis for our work, and we covered them in detail in Chapter 2.

Recent research on the Hofmann-Jost analysis has sought to provide bounds for programs written in richer languages. The ARTHUR analysis (Jost, 2004a; Jost, 2008) adds higher-order functions, where the major difficulty is that if a function is used several times, then the captured variables from its definition could be used several times (with the same type) and therefore so will the potential assigned to them. Multiple uses of the same potential could lead to an underestimate of the memory used. Thus Jost

adds constraints preventing the assignment of potential to captured variables. Inference is otherwise similar to before.

The RAJA type system (Hofmann and Jost, 2006) targets a language with assignment and object-oriented programming features based on Featherweight Java with updates. The types are now annotated by *views*, where each view maps classes to potential for that particular reference, and assigns a view to each of the fields of the object. The total potential is thus the sum of the potential from these views for each reference over every *access path* (chain of references) to it. To safely allow updates to references, views are given two kinds of annotations: the first gives the assignment of potential to the variable as usual; the second annotation describes an upper bound on the potential for *all* aliases of the data structure. Thus on examining a data structure we may use the first amount of potential, but on changing it we supply the second. For the object-oriented features conditions are imposed on subtyping so that inheritance and downcasting do not violate the given bounds.

The operational semantics in this work differ slightly from their earlier work, and the semantics presented in Chapter 2. Instead of using the benign sharing conditions, they avoid the problem by marking deallocated cells as invalid when they are no longer required.

However, only a type system and its soundness are presented for RAJA. The extra complexity would require more involved inference, especially for the object oriented features. Thus we can use the system to prove a bound, but not to produce one. It seems likely that a version of the system with assignment but not object-orientation could be significantly easier to develop an inference procedure for.

The Hofmann-Jost analysis has also been adapted to the Hume programming language as part of the Embounded project (Jost et al., 2007a). The project aims to certify the resource usage of real-time embedded programs written in Hume. The heap analysis incorporates the higher-order functions extension from ARTHUR and the researchers are experimenting with improvements to allow potential to be assigned to captured variables where a bound on the number of uses of the function can be established, and to assign potential to floating point values in proportion to their magnitude with the aim of obtaining more precise bounds on image processing algorithms (Jost, 2007).

Their implementation has also been extended to provide analyses of linear bounds on worst case execution time and stack space (Jost et al., 2007b). The stack space extension is roughly equivalent to the direct adaption we presented in Chapter 4, and

we anticipate that there would be little difficulty in using the more precise analyses from this thesis in their setting.

Recently work has begun in Nijmegen on the AHA project to develop a Hofmann-Jost-like amortized analysis which targets a lazy programming language and can infer some super-linear bounds (van Eekelen et al., 2007). They intend to investigate combining an amortized analysis with sized types (which we describe in detail below), but the work is at an early stage at the time of writing, and so it is unclear what form the results will take.

8.2 Sized types

The amortized analyses presented in this thesis produce bounds given in terms of the sizes of the input arguments, but the inference of these bounds never deals explicitly with data structure sizes, only the per-element (or per-layer in the case of the depth analysis) amounts of potential. We now consider the body of work based around the notion of *sized types*, where types are annotated with constraint variables or expressions which represent the size of the data structure or an upper bound on the size. Resource bounds can then be expressed in terms of these sizes and they are also useful for other purposes, such as optimisation.

An early use of a form of sized types is Reistad and Gifford's system to obtain execution time bounds and to facilitate dynamic parallelisation (Reistad and Gifford, 1994). The time bound is compared to the cost of spawning a new thread to decide whether parallel computation would present an overall improvement. There is a trade off between using an upper bound and unnecessarily spawning threads, and a lower bound where parallelisation opportunities may be missed. They use upper bounds.

They avoid the difficult task of inferring precise sizes and costs involved in recursively defined functions by providing some primitive functions with known size effects and cost, such as list `map` and `fold`. For other functions with non-trivial costs or size effects there is a special unbounded size, `long`, which is used when no more precise value can be inferred by the constraint solving.

Providing known primitives rather than using a more complex method ensures that sufficiently precise analyses can be made to allow dynamic parallelisation of a useful range of programs with an analysis of modest complexity (the constraints produced are solved by a quadratic fixed-point algorithm). Moreover, in some situations where no static information is available about the size of a data structure, the analysis may still

be able to obtain an execution time bound *relative* to the unknown size, and we can thus add runtime parallelisation based on the actual size observed.

Pareto et al. have studied the use of sized types to provide safety properties. They first considered showing productivity and evaluation in finite space of 'reactive systems' written in a functional programming language with unbounded streams for input and output (Hughes et al., 1996). One interesting aspect of dealing with streams is that the usual meaning of a sized type is reversed — a size annotation on a stream is a *lower* bound on its size.

Hughes and Pareto went on to study enforcing space bounds in a first-order 'embedded ML' language with region memory management (Hughes and Pareto, 1999). Effects are used to track changes in stack and region allocation, as well as to enforce memory safety. In both systems function signatures must be provided; there is no full type inference. Type checking for both systems relies on producing constraints in the form of Presburger formulae, which can be solved with the Omega calculator. More details about these two systems can be found in Pareto's thesis (Pareto, 2000).

Another strand of work begins with Chin and Khoo's paper on inferring sized types (Chin and Khoo, 2001). Again, the type system uses constraints expressed as Presburger formulae, but there is also a procedure to attempt to infer sizes where recursively defined functions are involved. First, a Presburger formula relating the size of the original parameters to the size of the arguments to the recursive calls is generated from the typing rules, then the transitive closure operation from the Omega calculator is used to attempt to find a fixed-point. Additional approximation heuristics are used to simplify the formula if a fixed-point cannot be found at the first attempt, for example because an accurate solution would not be expressible as a Presburger formula. At worst, this may also fail and yield no information about the sizes other than their existence (the resulting formula is just true).

The results of Chin and Khoo's analysis can be quite precise. For instance, consider this example from the paper which appends `xs` to itself iff `b` is true,

```
f b xs = case b of False -> xs
                 | True  -> append xs xs
```

where `append` is the usual list append function. In this type system booleans are given a 'size' of 0 for `False` and 1 for `True`. Together with disjunction the type system can infer a constraint giving an *exact* size expression for the result of `f` by reproducing the

case split in the formula:

$$\texttt{f} : \mathsf{Bool}^i \to \mathsf{list}^j \to \mathsf{list}^k, (i = 0 \wedge j \geq 0 \wedge k = j) \vee (i = 1 \wedge j \geq 0 \wedge k = j + j).$$

(Note that constraints using disjunctions in this way can also be used to provide maxima.) Thus another view of the sized types in Chin and Khoo's work is as an inference for a 'lightweight' form of dependent types — such as Xi's DML (Xi, 1998). DML also restricted the constraints to Presburger formulae to allow the checking of programmer supplied types for verification and optimisation purposes. Chin and Khoo infer suitable types instead of verifying ones supplied by the user.

Later work in this vein covers properties of the contents of collections (Chin et al., 2003), a type system for sized types for a language with references (Chin et al., 2005a) and a system for *checking* that specified memory bounds are met in an object-oriented language with explicit deallocation (Chin et al., 2005b). In the latter heap memory is tracked by a set of class name and size expression pairs, denoting how many of each is sufficient to evaluate an expression (when it appears in the assumptions of a judgement), or a lower bound on the number free afterwards (as part of the result's type information).

Vasconcelos has considered applying a size-types analysis in the style of Chin and Khoo to programs written in the Hume programming language (Vasconcelos, 2008). He addresses some problems in the soundness proof of the size types system, and then uses the information from the sized-types analysis in constructing a cost analysis for heap and stack space.

At this point we pause to consider how our notion of assigning potential via type annotations compares to sized types when extra features are added to the language. We noted while considering Jost's higher-order function extension that we must take care to either not assign potential to captured variables, or we must split it between all the uses of the function. With sized types the size of a captured variable does not depend on how many times the function is used, and no special effort is required. (The size may still appear in constraints or memory bounds obtained for evaluating the function, however.)

For references and assignment the situation is reversed: with potential, Jost's RAJA allows as many aliases as you like, and just keeps track of the total potential required for any assignment; whereas (Chin et al., 2005a) requires that the reference used for the assignment must be the *only* one which we attach size information to, because an update may invalidate the sizes of all the other references.

There are also differences in the form of bounds that can be expressed. In particular, for a nested structure such as a list of lists our analyses can assign potential (and hence infer bounds) in terms of the total number of elements in all of the inner lists. In a straightforward sized types system we can only express a *single* upper bound on the sizes of *all* the inner lists. That is, the type has the form

$$(\alpha\ \mathsf{list}^n)\ \mathsf{list}^m,$$

where m is a bound on the outer list and n the bound on the size of *every* inner list. Thus the best bound on the total number of elements is $m \times n$, even if only one inner list has n elements and the rest are empty. To express bounds on the total number of elements in a sized types system the user needs to be able to define a new size function for the list of lists. (Note that the depth system of Chapter 6 has a limited form of user-defined size function in the structure of the constructor's signature. For example, we can define trees with potential proportional to their total size, or proportional to their depth.)

On a related note, recent work by Chin et al. studies automated verification using sized types and separation logic, where user defined predicates can be asserted about data structures (Chin et al., 2007). Specifications for functions can use these predicates, which combine the size and separation information. This is powerful enough to specify invariants such as sorted lists and balanced trees. Techniques similar to those of their previous type systems are used to verify the specifications given to functions. The newly introduced separation conditions are transformed into arithmetic formulae which soundly approximate the desired conditions.

A rather different approach to calculating costs from sized types is via recurrence relations. Grobauer has used recurrence relation solving for computing execution time bounds for a DML program (Grobauer, 2001). Of course, starting from a DML program means that the sizes have already been computed, and must be linear because they are expressed in DML.

This leads us on to Vasconcelos and Hammond's analysis, which also uses recurrence relations to deal with recursive functions combined with an algorithm based on Reistad and Gifford, and which computes sizes as well as execution time (Vasconcelos and Hammond, 2004). The class of costs that can be inferred and the complexity of the analysis is dependent upon the recurrence relation solver. Thus to characterise the programs that these analyses would work well on we could start by examining which classes of recurrence relations have efficient algorithms to compute solutions (or close

approximations to solutions). Later work considers adding intersection types to the type system to allow functions to take multiple types (Simões et al., 2007). This is particularly important for higher-order functions where the supplied function argument may determine the size relationship of other arguments to the result.

8.3 Crary and Weirich

In (Crary and Weirich, 2000), the authors present a type system for checking execution time specified using a form of dependent typing. Rather than using a fixed notion of size, the programmer can explicitly define an abstract structure for a datatype and write a cost function for that structure. For instance, the paper describes representing binary trees by giving the tree structure alone without the contents, and a (type-level) function which computes a cost proportional to the size of the tree. Then term-level functions can be given types which assert, for instance, that the structure of the result is the same as that of the argument and the function has the cost given by the type-level cost function.

Type checking for this system is decidable, and includes verifying the bounds. Inference of the dependent types and bounds is not provided. One interesting aspect of this work is that they have constructed a compiler which produces code in a variant of their Typed Assembly Language (TAL) with a virtual register acting as a clock — the compiler transforms the type information about time bounds when producing TAL code, certifying the runtime cost of the assembler code.

8.4 Resource bounds for logic programs

Debray and Lin have studied the analysis of execution time for logic programs (Debray and Lin, 1993). In contrast to the previous analyses the language is untyped, but the basic notions are the same: some size measure is chosen for variables in the program (in fact, they even suggest determining the types to decide which size measure to use), relationships between the sizes of variables are derived from the program structure, and recursion is dealt with by solving recurrence relations. However, we also need some way to deal with backtracking. Thus the analysis also features techniques for bounding the number of solutions of each clause, and combines this with the size analysis to bound the evaluation time.

These techniques can be adapted to find *lower* bounds on execution time (Debray et al., 1997). As with Reistad and Gifford, this has applications in automated parallelisation. This size inference, along with other static analyses used to facilitate it, has been integrated into the CiaoPP abstract interpretation static analysis system; moreover, the abstractions produced using it have been suggested as certification of execution time for a Proof Carrying Code system (Hermenegildo et al., 2005).

We note two further pieces of follow on work; (Navas et al., 2007) considers bounding the usage of user specified resources such as open files and network messages sent. (Our amortized analyses could be adapted similarly, see Section 9.1.5.)

Finally, their techniques have been applied directly to Java bytecode (Albert et al., 2007). The idea is to reconstruct enough program structure to perform the usual analysis. Dynamic dispatch is handled by considering all of the methods which could be invoked (as if it were some non-deterministic choice) and alias handling appears to be delegated to a separate analysis. Of course, the number of solutions analysis is no longer necessary.

8.5 Quasi-interpretations

As we discussed in Section 2.2, our amortized analyses have their roots in the study of languages which capture particular complexity classes. Similarly, quasi-interpretations were initially used to provide complexity results, but are now also used in more practical situations. A quasi-interpretation is an assignment to each constructor and function (and by extension all terms) of a mapping from argument sizes to a bound on the size of the result produced. It must also satisfy a monotonicity property.

Our starting point for considering quasi-interpretations is Amadio's work on *max-plus quasi-interpretations* (Amadio, 2005). It is particularly interesting for two reasons: the form of quasi-interpretations used, and the results on the synthesis of such interpretations (which amounts to finding bounds on the sizes of values in the program). The form involved is max-plus polynomials, where max takes the role of the additive operator in the polynomial, and plus the multiplicative operator. They arose in the examination of discrete event systems, where the maximum time is taken whenever part of a system waits for several concurrent events, and times are added whenever several events must happen sequentially. Amadio uses such max-plus polynomials to form quasi-interpretations.

Amadio proves that the synthesis problem for max-plus quasi-interpretations is

NP-hard, but also shows that if we restrict ourselves further to *multilinear* max-plus polynomials, which are those of the form

$$\max_{I \subseteq \{1,\ldots,n\}} \left(\sum_{i \in I} x_i + a_I \right),$$

for argument sizes x_i and constants a_I, then synthesis is NP-complete and gives a suitable algorithm. For comparison, the depth type system of Chapter 6 gives bounds equivalent to

$$\max_{I \subseteq \{1,\ldots,n\}} \left(\sum_{i \in I} a_{I,i} x_i + a_I \right),$$

but does not directly bound the size of values, only the memory usage.

An interesting result from the paper is that programs in a simplified version of Hofmann's non-size-increasing language, LFPL, (briefly introduced on page 13 and which uses the $\diamond$ types and values to control allocation) can be given max-plus quasi-interpretations derived from the positions of $\diamond$ types.

Further work on quasi-interpretations applies the technique in a simple language with cooperative multithreading to bound the amount of heap memory used in each thread between yields (Amadio and Dal Zilio, 2006). This is combined with a termination analysis to ensure liveness, and as a result can also provide what they describe as 'rather rough' stack bounds because the termination analysis bounds the stack depth and the quasi-interpretations bound the frame size.

Recent work in the area concentrates on generalising quasi-interpretations to *sup-interpretations* (Marion and Péchoux, 2006), which can be applied to a wider range of programs. However, no results on the synthesis of these has been published at the time of writing.

8.6 Profiling and Symbolic Evaluation

A quite different approach for investigating resource usage is to collect information about the behaviour of the program during execution. There is a long tradition of using profiling tools to find the sections of programs responsible for excessive execution times. An excellent example is Knuth's description of a study of Fortran programs (Knuth, 1971). The technique was so successful (after profiling the profiler they were able to double its speed with less than an hour's work) that Knuth coined the term *profile* and strongly advocated the use of profiling.

Tools for memory profiling are also common, and have been exceptionally useful in two functional programming settings. First, it can be difficult for programmers to predict the memory usage of their code in the presence of lazy evaluation. Runciman et al. have investigated the use of profiling on programs written in Lazy ML (Runciman and Wakeling, 1993) and subsequently Haskell (Röjemo and Runciman, 1996), showing that it can reveal parts of programs where refactoring the code to make it lazier can streamline memory usage, and parts where making it more eager can dispose of a large data structure earlier. They also found profiling useful for finding bugs and potential improvements in the compiler as well as the program being examined.

The second setting is described in the retrospective on region memory management in the ML Kit (Tofte et al., 2004). Experimentation with their profiler led to the conclusion that a slightly different style of programming from normal would help to reduce memory usage with regions, and advocate this style and the use of profiling in the development process to support it in the Kit's user manual (Tofte et al., 2006).

In comparison to static analysis of resource usage, profiling yields more precise information, but only for the range of test cases used. Thus they can be used in a complementary fashion to investigate and debug actual program behaviour on test cases, while using a static analysis to obtain worst-case bounds.

A compromise between profiling and static analysis is the symbolic evaluation approach of Liu et al. (Unnikrishnan et al., 2000) which follows on from their time analysis, (Liu and Gòmez, 2001), which in turn is based on (Rosendahl, 1989). They add profiling code to the original program, then further transform it to handle 'partially known' input—for example, a list consisting of some number of special 'unknown' values. This new program attempts to compute all branches of a conditional where it depends upon an unknown value, and so also handles values which are the combination of several results from different branches.

While this approach can also yield some precise bounds, it presents several difficulties: the transformed program must be rerun on partial data structures of every size we are interested in; it may not terminate, even if the original program would; and the runtime of the transformed program often substantially exceeds the runtime of the original.

Chapter 9

Further work

We split our consideration of further work that could be undertaken on our analyses into two sections. First, we discuss the areas that would be applicable to any of the analyses. We follow this by a section on future work specifically for the depth analysis, because it is quite different from our other systems, and presents more opportunities for improvement.

9.1 General topics

9.1.1 Language features

It is natural to wish the analyses to cope with richer programming languages, and we have already mentioned the work that Jost et al. have been pursuing in Section 8.1. We conjecture that their mechanisms for higher-order functions and assignment could be adapted to our stack space analyses without difficulty.

We might also wish to avoid the requirement for programs to be presented in let-normal form (see page 4). While this can be easily achieved by a source code transformation it would also be possible to treat the other syntactic forms as derived expressions. For example,

$$\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \quad := \quad \mathsf{let}\ x = e_1\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3,$$

or use such derived forms to produced derived typing rules. Another direction might be to model the unspecified evaluation order present in many language specifications by requiring any order to be valid. For instance, we might allow for the order of evaluating the two subexpressions in a pair to be unspecified by adding constraints to allow either to be evaluated first (using a system like plain Hofmann-Jost for illustrative purposes):

$$\frac{\Gamma_1, n_1 \vdash_{\Sigma,F} e_1 : T_1, n_1' \,|\, \Phi_1 \qquad \Gamma_2, n_2 \vdash_{\Sigma,F} e_2 : T_2, n_2' \,|\, \Phi_2 \qquad \Phi = \Phi_1 \cup \Phi_2 \cup \left\{ \begin{array}{lll} n \geq n_1, & n - n_1 + n_1' \geq n_2, & \\ n \geq n_2, & n - n_2 + n_2' \geq n_1, & n - n_1 + n_1' - n_2 + n_2' \geq n' \end{array} \right\}}{\Gamma_1, \Gamma_2, n \vdash_{\Sigma,F} (e_1, e_2) : T_1 \otimes T_2, n' \,|\, \Phi} \text{(PAIR}')$$

Polymorphism is also desirable, but presents more difficulties. For the simpler case of a first-order language, the only way to use a variable of an unknown type is by contraction. We might thus assign a 'symbolic' constraint variable to each type variable and then when instantiating the function at a particular type replace these variables and the constraints on them produced by contraction by a copy for each set of concrete annotations. For example, if we have the pairing function with signature

$$\alpha(k_1), n \to \alpha(k_2) \otimes \alpha(k_3), n' \,|\, \{k_1 = k_2 + k_3,\ n \geq n'\},$$

for some type variable α, then use it with a sum type $(\mathsf{bool}, k_b) + (\mathsf{int}, k_i)$, we get

$$(\mathsf{bool}, k_b) + (\mathsf{int}, k_i), n \to ((\mathsf{bool}, k_{b,1}) + (\mathsf{int}, k_{i,1})) \otimes ((\mathsf{bool}, k_{b,2}) + (\mathsf{int}, k_{i,2})), n' \,| \\ \{k_b = k_{b,1} + k_{b,2},\ k_i = k_{i,1} + k_{i,2},\ n \geq n'\},$$

copying the k_1 constraint once for each of the annotations, k_b and k_i.

For higher-order types it is not clear how to deal with applying functions which have unknown constraints relating the potential of its arguments to its result. One alternative would be to follow the sized types work in (Simões et al., 2007) and investigate using intersection types.

9.1.2 Integration with safety analyses

We have presumed throughout the presence of some memory safety analysis, although we have largely ignored it. However, the use of the separation information provided by such an analysis in the system of Chapter 5 suggests that greater integration may be beneficial. Indeed, it is likely that a weaker condition about data flow from the safety analysis could replace the separation condition in that system, which may improve some of the bounds produced. Also, the layering proposal we will discuss in Section 9.2.1 is reminiscent of Konečný's DEEL safety analysis (Konečný, 2003), so it may be useful to combine the two.

9.1.3 Accumulating parameters

We noted in Section 4.3 that accumulating parameters could be a source of imprecision, albeit an uncommon one due to tail call optimisation. Nevertheless, such parameters are certain to appear in some non-tail-recursive functions (for example, because it happens to produce a list in the desired order), and it would be preferable to deal with the problem. We suggest that it may be possible to 'borrow' potential that we wish to assign to a newly allocated cell, and thus both assign it to the accumulator and to a subsequent function call. To make such borrowing sound, we conjecture that we could make further use of information from the memory safety analysis to show that the accumulator will not examined until after the recursion terminates.

9.1.4 Using information from the linear programs

It is possible to extract some additional information from the constraints generated by the analyses. An existing example features in Jost's implementation of the original heap space analysis (Jost, 2004b): it checks the solutions against the constraints to detect points in the program where the analysis 'leaks' potential — that is, where it may overestimate because it loses track of some free memory. Some of these may correspond to memory leaks in the program. This could be applied to our analyses, although the system with max in Chapter 6 may produce too many false positives to be useful.

Another avenue for investigation would to be present better information when the linear program is infeasible and so no bound can be obtained. Some initial experimentation suggests that if we add extra 'slack' potential to every function call and minimise this slack, then the locations where some slack remains are likely to be the parts preventing the analysis from finding a bound (for instance, because they require super-linear amounts of memory). This process is similar to the initial phases of linear programming using the Simplex method.

Finally, we might seek to simplify the generated linear programs so as to present them as part of the feedback to the user, or perhaps to speed up the analysis of a large program, particularly where resource polymorphism is involved.

9.1.5 Other resources

It would be desirable to extend our techniques to analyse other resources which are likely to have linear bounds. This should be straightforward in our amortized analyses by setting $\text{size}(c) = \text{stack}(f) = 0$ for all c, f and instead putting requirements for potential on the functions which open and close files, send messages and so on. Note that the depth analysis would require the resources to obey a stack discipline, or be modified as suggested in Section 9.2.3.

9.1.6 Region memory management

Tofte and Talpin's region memory management system (Tofte and Talpin, 1997) allows memory to be efficiently allocated and deallocated by placing data in lexically scoped *regions*. While these regions are introduced and destroyed according to a stack discipline, their maximum size is not known in advance and so large regions are usually made up of several non-contiguous sections of memory, resulting in some wastage. If a Hofmann-Jost style system could bound the size of each region at its point of introduction (in terms of the sizes of the live variables at that point), then we could allocate the regions in a single stack throughout, possibly reducing some of this wastage and simplifying allocation. Additionally, it should be possible to then develop an analysis to give bounds on the whole program, for which techniques from our stack analyses are likely to provide some assistance.

9.2 Further work for the depth analysis

9.2.1 Layering to separate contents from container structure

In Section 6.1 we noted that the potential functions we define for nested datatypes calculate the highest cost path from root of the structure to the innermost parts, weighted by the annotations. However, this had an adverse effect on some of the examples — even just swapping two values in different levels of a binary tree can require extra potential and so inflate the memory bounds (see Example 6.15). It also part of the reason why the plus-contraction scales all of the annotations in a type uniformly rather than independently, as in our previous systems (see the discussion on page 103 for details). Instead we would like to calculate the potential as the sum of the depth of the tree and the maximum potential assigned to an element of the contents — essentially slicing

the data structure into layers, whose potential is summed. We choose this informal definition because it corresponds to typical stack usage: we recursively process the data structure, dealing with one element of contents at a time, *and* expect to be able to rearrange the contents within the structure.

The difficulty with realising this is that we can no longer unfold a 'layered' data structure locally in our structured contexts. The structure and contents should collect in different parts of the context so that we obtain the correct potential. As an alternative, we conjecture that we can solve this problem by introducing a contents marker context former, $\Gamma^\dagger$, to show that Γ should be counted separately from the 'structure' surrounding it, and another context former, $[\Gamma]$ to delineate the extent of the unfolded data structure. Marked 'contents' would thus be counted as if it were in a single max bunch at the enclosing $[\cdot]$. The type system would have extra rules to allow the contents to move around the unfolded structure as we like.

Some preliminary work have been done on a similar (but simpler) construct which sums the potential from all of the contents. Thus we expect that a sound type system can be built on this basis. However, it is not clear how to adapt the inference process because we can no longer expand every context to an equivalent max-of-sums form. Nevertheless, we would hope that the extras user input would be at most marking the 'contents' in constructor and function signatures.

9.2.2 Reducing expansion

It would be useful to reduce the amount of expansion required in the inference process of Chapter 7, for three reasons: it may give us insight as to an inference process for a layered system; it could remove the potential for exponential blow-up; and the resulting typings would be easier for users to follow. A possible approach to this would be to look for the variables currently of interest in the context, and perform a local expansion only for those parts.

9.2.3 Heap space bounds with maxima

The analysis from Chapters 6 and 7 could provide some benefits for heap space bounds. We have already seen in Example 6.14 that we can obtain some bounds that Hofmann-Jost does not allow, and that we can obtain total space bounds by making the constructor signatures entirely additive. One point to investigate is whether requiring entirely additive constructor signatures would allow a more relaxed version of plus-contraction,

bridging the gap between the depth system and Hofmann-Jost. The major obstacle to a heap analysis, however, is that the 'local' context change in the D-LET rule is only sound under a stack discipline, as we described on page 103.

A method for tackling this would be to try replacing the D-LET rule with one which requires the first subexpression's context to be separate from the rest, ensuring that no maxima are broken by allocation:

$$\frac{\Gamma \vdash^{g,\mathsf{false}}_{\Sigma,F} e_1 : T_0, k_0 \,|\, \Phi_1 \qquad \Delta, x : T_0, k_0 \vdash^{g,t}_{\Sigma,F} e_2 : T, k' \,|\, \Phi_2}{\Gamma, \Delta \vdash^{g,t}_{\Sigma,F} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T, k' \,|\, \Phi_1 \cup \Phi_2} \quad \text{(D-LET}')$$

(In the usual rule, Γ may appear in the middle of Δ's structure, see page 100.) However, this may go too far, especially when e_1 does not perform any allocation, and the effect on inference is currently unknown.

9.2.4 Inference of function signature structure

We assumed that the user supplies the function signature structures to simplify the construction of our inference process in Chapter 7. While this only requires a small amount of user input, it would be preferable to make the inference entirely automatic.

We could attempt this by introducing placeholders consisting of a set of variable name and type pairs in the 'desired' contexts of the inference process, which would mark a place where some subcontext is required for a function call, but the precise structure is not known. Then as the inference progresses we can attempt to fill in some of the structure once we know how the variables in the placeholder are bound. Again, this may prevent us fully expanding contexts during the inference process, so work on reducing expansion may help.

9.2.5 Logarithmic bounds and invariants

We have given an analysis which is capable of giving stack space bounds with respect to the depth of data structures, and tree structures with logarithmic depth are often used for efficient data structures. Most of the functions used for processing these will thus use at most logarithmic stack space with respect to the data's total size.

However, the tree structures are usually built from flat input, either in the form of a single list, or by accumulating elements one at a time. While we can infer stack space bounds proportional to the resulting tree's depth, our analysis cannot show that this is equivalent to a logarithmic bound in terms of the input's size.

Inferring logarithmic bounds presents three challenges: representing logarithmic amounts of potential assigned to the input (especially where the input is added to the tree incrementally); relating the logarithmic potential to the depth of the tree; and showing any invariants required for that relationship.

The heap sort program in Appendix A gives a special case where this is relatively easy. We could assign a logarithmic amount of potential to the input list as a whole, transfer it to the list length when that is computed, then 'release' constant amounts of potential each time the list length is halved. This could work because the code halves the list length every time it adds a layer to the tree. This trick does not generalise, however. For example, a simpler version of heap sort adds the elements to the tree incrementally, so there is no explicit list length that could be used.

Chapter 10

Conclusions

The core thesis of this work was that we could construct good type-based amortized analyses of programs' stack space usage where that usage is linear in terms of the input's size. We have presented several such analyses, and in particular have tailored the two in Chapter 5 and Chapters 6 and 7 specifically for the behaviour of stack space allocation, achieving good bounds.

In more detail, this work has presented the following contributions:

- a Continuation Passing Style transformation which replaces stack space usage by heap space usage, and thus allows us to apply a heap space analysis to get a bound on the transformed program, and also to get a bound on the total memory or stack memory usage of the *original* program; but we then showed that the transformation introduces a requirement for certain types to be equal which severely limits such type based analyses on CPS transformed programs (Chapter 3);

- we showed that directly adapting the Hofmann-Jost heap space analysis to include stack space yields a usable stack space analysis for bounds in terms of total data structure size (Chapter 4), and that extending the form of the post-evaluation bounds to include the sizes of arguments as well as the size of the result provides us with a better analysis that is well-suited to producing stack space bounds (Chapter 5);

- we showed that stack space bounds expressed with respect to the *depth* of the input is practical using these techniques, by a novel use of extra structure in the typing contexts to represent the form of the bounds, and presented a inference algorithm for this extra structure (Chapters 6 and 7).

We have supported these contributions by providing correctness and resource simulation proofs for the Continuation Passing Style transformation, and soundness proofs for the type systems which show that the predicted stack usage is a strict upper bound on the real usage. We have also extended the soundness results to non-terminating programs. All of the analyses in this thesis have been implemented and tested on examples. Sample results are shown in Appendix A.

Appendix A

Functional Heap Sort Example

We demonstrate the implementations of our analyses on a functional heap sort program. The heap sort program is based on the Standard ML example from (Paulson, 1996, Section 4.16), rewritten in LFD. We first present the code itself with a few comments, and then proceed to examine the results of each analysis in turn.

The program begins with unannotated type definitions and function signatures. The size and stack measures are defined here by the numbers in `(*n*)`. Here we estimate the size measure by counting the number of values involved in each declaration — a reasonable amount if each value is represented by one machine word. Similarly, we estimate the stack measure by counting the maximum number of simultaneously live variables in the function body.

```
type intlist = Nil(*0*) | Cons(*2*) of int, intlist
type inttree = Leaf(*0*) | Node(*3*) of inttree, inttree, int

val siftdown(*7*): inttree, inttree, int -> inttree
val heapify(*4*): int, intlist -> inttree * intlist
val addlength(*2*): intlist, int -> int
val length(*1*): intlist -> int
val fromList(*2*): intlist -> inttree

val leftrem(*4*): inttree -> int * inttree
val delmin(*3*): inttree -> inttree
val toList(*2*): inttree, intlist -> intlist

val sort(*2*): intlist -> intlist
```

The code follows, written in the explicit let-normal form LFD requires (see Section 2.1):

```
let siftdown(t1, t2, w) =
  let l = Leaf in
  match t1 with
    Leaf -> Node(l,l,w)
  | Node(t11,t12,v) ->
      (match t2 with
        Leaf ->
          if v < w then let a = Node(l,l,v) in Node(a,l,w)
                   else let a = Node(l,l,w) in Node(a,l,v)
      | Node(t21,t22,u) ->
          if u < w & v < w then
            let a = Node(t11,t12,v) in
            let b = Node(t21,t22,u) in
              Node(a,b,w)
          else if w < u & v < u then
            let a = Node(t11,t12,v) in
            let b = siftdown(t21,t22,w) in
              Node(a,b,u)
          else
            let a = siftdown(t11,t12,w) in
            let b = Node(t21,t22,u) in
              Node(a,b,v)
      )


let heapify(i, l) =
  if i = 0 then let lf = Leaf in (lf, l)
  else
    match l with
      Cons(v,vs) ->
        let tl1 = heapify(i/2, vs) in
        match tl1 with (t1, vs1) ->
          let tl2 = heapify((i-1)/2, vs1) in
          match tl2 with (t2, vs2) ->
            let t3 = siftdown(t1, t2, v) in
            (t3, vs2)

let addlength(l, n) =
  match l with
    Nil -> n
  | Cons(_,vs)' -> let n' = n+1 in addlength(vs, n')

let length l = addlength(l, 0)
```

```
let fromList vs =
  let len = length vs in
  let tl = heapify(len, vs) in
  match tl with (t, _) -> t


let leftrem t =
  match t with
    Node(t1,t2,v) ->
      (match t1 with Leaf ->
        (* t2 must be Leaf too *)
        let lf = Leaf in (v,lf)

      | Node(_,_,_)' ->
        let wt' = leftrem t1 in
        match wt' with (w,t') ->
          let t'' = Node(t2,t',v) in
          (w,t'')
      )
```

The `delmin` function illustrates the need for inexhaustive pattern matching; callers are responsible for ensuring that the tree is not empty. While the program could be rewritten to eliminate the inexhaustive match, it is more convenient to analyse programs as they were written. In particular, the practice of adding 'dummy' cases must be avoided because they can affect the precision of the analysis. For example, if we were to add a dummy `Leaf -> Leaf` pattern here then our analyses would be unable to show that the heap space for the destroyed node is returned because no deallocation takes place in the dummy case.

```
let delmin t =
  match t with
    Node(t1,t2,_) ->
      (match t1 with
        Leaf -> Leaf
      | Node(_,_,_)' ->
          let wt = leftrem t1 in
          match wt with (w,t') ->
            siftdown(t2, t', w)
      )

let toList (t, a) =
```

```
  match t with
    Leaf -> a
  | Node(_,_,v)' -> let t' = delmin t in
                    let a' = Cons(v,a) in
                    toList (t', a')

let sort l =
  let t = fromList l in
  let n = Nil in
  toList (t, n)
```

If we determine the stack usage of `sort` on a list l by examining the code by hand we find that it is determined by the following chain of calls:

$$\text{stack}(\texttt{sort}) + \text{stack}(\texttt{fromList}) + \text{stack}(\texttt{heapify}) + \text{stack}(\texttt{siftdown}) \times (\log_2(|l|+1)+1),$$

which is equal to $15 + 7 \times (\log_2(|l|+1))$. If we measure the total space usage then the deallocated list elements can be used to offset some of the stack space required before the corresponding tree node is created, giving a bound of $15 + 5 \times (\log_2(|l|+1))$.

A.1 The Hofmann-Jost heap analysis

Applying the version of the Hofmann-Jost heap space analysis presented in Chapter 2 yields the following type signatures:

```
siftdown   : 3, inttree[0|#,#,int,0], inttree[0|#,#,int,0], int
               -> inttree[0|#,#,int,0], 0;
siftdown   : 3, inttree[0|#,#,int,0], inttree[0|#,#,int,0], int
               -> inttree[0|#,#,int,0], 0;
heapify    : 0, int, intlist[0|int,#,1]
               -> inttree[0|#,#,int,0] * intlist[0|int,#,1], 0;
addlength  : 0, intlist[0|int,#,0], int -> int, 0;
length     : 0, intlist[0|int,#,0] -> int, 0;
fromList   : 0, intlist[0|int,#,1] -> inttree[0|#,#,int,0], 0;
leftrem    : 0, inttree[0|#,#,int,0] -> int * inttree[0|#,#,int,0], 3;
delmin     : 0, inttree[0|#,#,int,0] -> inttree[0|#,#,int,0], 3;
toList     : 0, inttree[0|#,#,int,0], intlist[0|int,#,1]
               -> intlist[0|int,#,1], 0;
sort       : 0, intlist[0|int,#,1] -> intlist[0|int,#,1], 0;
```

The types in the signatures are given as a type name followed by a list of constructor signatures and annotations, separated by `|`s. For instance, `intlist[0|int,#,1]` is a

list of integers, where each `nil` has 0 units of potential and each `cons` has 1 unit of potential. The `#` in a signature represents a recursive appearance of the data type.

The `siftdown` function appears twice because of resource polymorphism; the analysis finds a solution for each of the uses of `siftdown` outside of its definition (in `heapify` and `delmin`). In this case, the best signature for both uses is the same.

These bounds agree with the general result given in Example 2.9, that `sort l` requires

$$|\texttt{l}| \times |\text{size}(\texttt{node}) - \text{size}(\texttt{cons})|$$

units of free heap space. Then, if we fix the size of the list elements to be the size of the heap nodes,

```
type intlist = Nil(*0*) | Cons(*3*) of int,intlist
```

we get an in-place sort:

```
siftdown   : 3, inttree[0|#,#,int,0], inttree[0|#,#,int,0], int
               -> inttree[0|#,#,int,0], 0;
siftdown   : 3, inttree[0|#,#,int,0], inttree[0|#,#,int,0], int
               -> inttree[0|#,#,int,0], 0;
heapify    : 0, int, intlist[0|int,#,0]
               -> inttree[0|#,#,int,0] * intlist[0|int,#,0], 0;
addlength  : 0, intlist[0|int,#,0], int -> int, 0;
length     : 0, intlist[0|int,#,0] -> int, 0;
fromList   : 0, intlist[0|int,#,0] -> inttree[0|#,#,int,0], 0;
leftrem    : 0, inttree[0|#,#,int,0] -> int * inttree[0|#,#,int,0], 3;
delmin     : 0, inttree[0|#,#,int,0] -> inttree[0|#,#,int,0], 3;
toList     : 0, inttree[0|#,#,int,0], intlist[0|int,#,0]
               -> intlist[0|int,#,0], 0;
sort       : 0, intlist[0|int,#,0] -> intlist[0|int,#,0], 0;
```

A.2 The CPS transformation

We now apply an implementation of the CPS transformation described in Chapter 3 before using the Hofmann-Jost heap analysis. Recall that this technique bounds the total memory usage *except* for the largest stack frame.

We present a sample of the resulting typing signatures:

```
c%2        : 12, inttree[0|#,#,int,1], int, inttree[0|#,#,int,1],
                 inttree[0|#,#,int,1], int,
```

```
                c%stack[inttree[0|#,#,int,1],#,0|
                  intlist[0|int,#,2],int,#,2|#,0|0|
                  int,inttree[0|#,#,int,1],inttree[0|#,#,int,1],int,#,2|
                  int,inttree[0|#,#,int,1],#,0|intlist[0|int,#,2],#,0|
                  int,inttree[0|#,#,int,1],#,0|int,int,#,0|#,0|
                  intlist[0|int,#,0],#,0|int,inttree[0|#,#,int,1],#,0]
              -> intlist[0|int,#,2], 6;
c%unwind__intlist: 0, intlist[0|int,#,2],
                c%stack[inttree[0|#,#,int,0],#,0|
                  intlist[0|int,#,0],int,#,0|#,0|0|
                  int,inttree[0|#,#,int,0],inttree[0|#,#,int,0],int,#,0|
                  int,inttree[0|#,#,int,0],#,0|intlist[0|int,#,0],#,0|
                  int,inttree[0|#,#,int,0],#,0|int,int,#,0|#,0|
                  intlist[0|int,#,0],#,0|int,inttree[0|#,#,int,0],#,0]
              -> intlist[0|int,#,2], 0;
sort        : 11, intlist[0|int,#,2],
                c%stack[inttree[0|#,#,int,1],#,0|
                  intlist[0|int,#,2],int,#,2|#,0|0|
                  int,inttree[0|#,#,int,1],inttree[0|#,#,int,1],int,#,2|
                  int,inttree[0|#,#,int,1],#,0|intlist[0|int,#,2],#,0|
                  int,inttree[0|#,#,int,1],#,0|int,int,#,0|#,0|
                  intlist[0|int,#,2],#,0|int,inttree[0|#,#,int,1],#,0]
              -> intlist[0|int,#,2], 6;
c%main     : 11, intlist[0|int,#,2] -> intlist[0|int,#,2], 6;
```

These show the four kinds of function in the transformed program: `c%2` is a closure generated while transforming the `siftdown` function; `c%unwind__intlist` is the stack unwinding function which is called when a function returns a value of type `intlist` and we need to determine which continuation to call; `sort` is the transformed version of the original function; and `c%main` is a wrapper function for `sort` introduced by the implementation which eliminates the stack. We can easily read the before and after free memory bounds from the type of the wrapper function, whereas the after bounds for most functions is obscured by the stack type. The interpretation of the bounds was presented in Example 3.13 on page 49.

By setting the original heap structure sizes to zero we can obtain a stack only bound, as described in the main text:

```
c%2        : 12, inttree[0|#,#,int,4], inttree[0|#,#,int,4], int,
                 inttree[0|#,#,int,4], int, c%stack[...]
             -> intlist[0|int,#,4], 6;
c%unwind__intlist: 0, intlist[0|int,#,4], c%stack[...]
             -> intlist[0|int,#,4], 0;
sort       : 11, intlist[0|int,#,4], c%stack[...]
```

```
                -> intlist[0|int,#,4], 6;
c%main      : 11, intlist[0|int,#,4] -> intlist[0|int,#,4], 6;
```

In Section 3.3 we showed that the CPS transformed program can be used to bound the original program's memory usage if the stack frame sizes are used in place of the real closure sizes. This yields slightly different results to the actual CPS program:

```
  c%main      : 14, intlist[0|int,#,5] -> intlist[0|int,#,5], 8;
```

The linear increase in the space bound is because our naïve estimates for the stack frame sizes above include variables which are only required for a call to another function. The CPS transform does not place these variables in the closures and can thus execute in less space. If a compiler implementation made a similar optimisation, then we could also justify reducing the stack frame size and obtain a tighter bound.

A.3 Direct adaption and the give-back analysis

We now come to the direct stack adaption of Hofmann-Jost presented in Chapter 4 and the subsequent give-back version of Chapter 5. Considering first the combined heap and stack bounds with the direct adaption we obtain the signatures:

```
siftdown    : 7, inttree[0|#,#,int,4], inttree[0|#,#,int,4], int
                -> inttree[0|#,#,int,4], 0;
siftdown    : 15, inttree[0|#,#,int,4], inttree[0|#,#,int,4], int
                -> inttree[0|#,#,int,4], 8;
heapify     : 7, int, intlist[0|int,#,5]
                -> inttree[0|#,#,int,4] * intlist[0|int,#,5], 7;
addlength   : 0, intlist[0|int,#,0], int -> int, 0;
length      : 2, intlist[0|int,#,0] -> int, 2;
fromList    : 11, intlist[0|int,#,5] -> inttree[0|#,#,int,4], 11;
leftrem     : 0, inttree[0|#,#,int,4] -> int * inttree[0|#,#,int,4], 7;
delmin      : 8, inttree[0|#,#,int,4] -> inttree[0|#,#,int,4], 15;
toList      : 11, inttree[0|#,#,int,4], intlist[0|int,#,5]
                -> intlist[0|int,#,5], 11;
sort        : 13, intlist[0|int,#,5] -> intlist[0|int,#,5], 13;
```

Adding the 2 units of space for the initial stack frame for `sort`, for a list l this gives us the bound $15 + 5 \times |l|$, the best we can expect from a linear analysis of total space.

As with the CPS transformation's results, the combined bounds can be misleading when heap and stack memory are not interchangeable, and so we give a stack bound alone:

```
siftdown   : 7, inttree[0|#,#,int,7], inttree[0|#,#,int,7], int
               -> inttree[0|#,#,int,7], 0;
siftdown   : 15, inttree[0|#,#,int,7], inttree[0|#,#,int,7], int
               -> inttree[0|#,#,int,7], 8;
heapify    : 7, int, intlist[0|int,#,7]
               -> inttree[0|#,#,int,7] * intlist[0|int,#,7], 7;
addlength  : 0, intlist[0|int,#,0], int -> int, 0;
length     : 2, intlist[0|int,#,0] -> int, 2;
fromList   : 11, intlist[0|int,#,7] -> inttree[0|#,#,int,7], 11;
leftrem    : 0, inttree[0|#,#,int,7] -> int * inttree[0|#,#,int,7], 7;
delmin     : 8, inttree[0|#,#,int,7] -> inttree[0|#,#,int,7], 15;
toList     : 11, inttree[0|#,#,int,7], intlist[0|int,#,7]
               -> intlist[0|int,#,7], 11;
sort       : 13, intlist[0|int,#,7] -> intlist[0|int,#,7], 13;
```

The `length` function used above is tail recursive and so uses a constant amount of stack space. Replacing it with the linear function

```
let length l =
  match l with
    Nil -> 0
  | Cons(_,vs)' -> let l' = length vs in (1 + l')
```

produces the overestimate noted in Example 4.7:

```
siftdown   : 7, inttree[0|#,#,int,7], inttree[0|#,#,int,7], int
               -> inttree[0|#,#,int,7], 0;
siftdown   : 15, inttree[0|#,#,int,7], inttree[0|#,#,int,7], int
               -> inttree[0|#,#,int,7], 8;
heapify    : 7, int, intlist[0|int,#,7]
               -> inttree[0|#,#,int,7] * intlist[0|int,#,7], 7;
length     : 0, intlist[0|int,#,1] -> int, 0;
fromList   : 11, intlist[0|int,#,8] -> inttree[0|#,#,int,7], 11;
leftrem    : 0, inttree[0|#,#,int,7] -> int * inttree[0|#,#,int,7], 7;
delmin     : 8, inttree[0|#,#,int,7] -> inttree[0|#,#,int,7], 15;
toList     : 11, inttree[0|#,#,int,7], intlist[0|int,#,7]
               -> intlist[0|int,#,7], 11;
sort       : 13, intlist[0|int,#,8] -> intlist[0|int,#,7], 13;
```

The give-back analysis solves this by reassigning the potential of the list argument back to the next use of the list (which is the call to `heapify`):

```
siftdown   : 7, inttree[0>0|#,#,int,7>0], inttree[0>0|#,#,int,7>0], int
               -> inttree[0>0|#,#,int,7>0], 0;
```

```
siftdown   : 15, inttree[0>0|#,#,int,7>0], inttree[0>0|#,#,int,7>0], int
               -> inttree[0>0|#,#,int,7>0], 8;
heapify    : 7, int, intlist[0>0|int,#,7>0]
               -> inttree[0>0|#,#,int,7>0] * intlist[0>0|int,#,7>0], 7;
length     : 0, intlist[0>0|int,#,1>1] -> int, 0;
fromList   : 11, intlist[0>0|int,#,7>0] -> inttree[0>0|#,#,int,7>0], 11;
leftrem    : 0, inttree[0>0|#,#,int,7>0] -> int * inttree[0>0|#,#,int,7>0], 7;
delmin     : 8, inttree[0>0|#,#,int,7>0] -> inttree[0>0|#,#,int,7>0], 15;
toList     : 11, inttree[0>0|#,#,int,7>0], intlist[0>0|int,#,7>0]
               -> intlist[0>0|int,#,7>0], 11;
sort       : 13, intlist[0>0|int,#,7>0] -> intlist[0>0|int,#,7>0], 13;
```

A.4 The depth type system

We now turn to the type system of Chapter 6 and accompanying inference in Chapter 7. Note that we adopt the change to `siftdown` discussed in Example 7.6 on page 147. Before we obtain stack space bounds with respect to depth, we first note that they can provide bounds on the stack space with respect to the total size of the arguments, too. To do this, we provide the inference with entirely additive signatures:

$$\Sigma = \left[\begin{array}{r} \texttt{Nil} \mapsto \forall k.() \to \texttt{intlist}(k) \\ \texttt{Cons} \mapsto \forall k.(\texttt{int},\texttt{intlist}(k),k) \to \texttt{intlist}(k) \\ \texttt{Leaf} \mapsto \forall k.() \to \texttt{inttree}(k) \\ \texttt{Node} \mapsto \forall k.(\texttt{inttree}(k),\texttt{inttree}(k),\texttt{int},k) \to \texttt{inttree}(k) \\ \texttt{siftdown} \mapsto (\texttt{inttree}(k_1),\texttt{inttree}(k_2),\texttt{int}(k_3),k_4) \to \texttt{inttree}(k_5),k_6 \\ \texttt{heapify} \mapsto (\texttt{intlist}(k_1),k_2) \to \texttt{inttree}(k_3) \otimes \texttt{intlist}(k_4),k_5 \\ \texttt{length} \mapsto (\texttt{intlist}(k_1),k_2) \to \texttt{int},k_3 \\ \texttt{fromList} \mapsto (\texttt{intlist}(k_1),k_2) \to \texttt{inttree}(k_3),k_4 \\ \texttt{leftrem} \mapsto (\texttt{inttree}(k_1),k_2) \to \texttt{int} \otimes \texttt{inttree}(k_3),k_4 \\ \texttt{delmin} \mapsto (\texttt{inttree}(k_1),k_2) \to \texttt{inttree}(k_3),k_4 \\ \texttt{toList} \mapsto (\texttt{inttree}(k_1),\texttt{intlist}(k_2),k_3) \to \texttt{inttree}(k_4),k_5 \\ \texttt{sort} \mapsto (\texttt{intlist}(k_1),k_2) \to \texttt{intlist}(k_3),k_4 \end{array}\right]$$

Note that the notation for the implementation of this analysis follows Chapter 7. Using these signatures, the implementation of the analysis yields the overall bound that we expect from the previous analyses:

```
siftdown: (t1:inttree(7),t2:inttree(7),w:int,7) -> inttree(7), 0
siftdown: (t1:inttree(7),t2:inttree(7),w:int,7) -> inttree(7), 0
```

```
heapify: (i:int,l:intlist(7),7) -> inttree(7)*intlist(7), 7
length: l:intlist(1) -> int, 0
fromList: (vs:intlist(7),11) -> inttree(7), 11
leftrem: (t:inttree(7),0) -> int*inttree(7), 7
delmin: (t:inttree(7),0) -> inttree(7), 7
toList: (t:inttree(7),a:intlist(7),13) -> intlist(7), 13
sort: (l:intlist(7),13) -> intlist(7), 13
```

Thus, once we add the initial stack frame for `sort`, we see that `sort l` can run in $7 \times |\mathtt{l}| + 15$ units of stack space.

For the analysis with respect to depth we use signatures that take the maximum potential where appropriate:

$$\Sigma = \left[\begin{array}{r} \mathtt{Nil} \mapsto \forall k.() \to \mathtt{intlist}(k) \\ \mathtt{Cons} \mapsto \forall k.(\{\mathtt{int};\mathtt{intlist}(k)\},k) \to \mathtt{intlist}(k) \\ \mathtt{Leaf} \mapsto \forall k.() \to \mathtt{inttree}(k) \\ \mathtt{Node} \mapsto \forall k.(\{\mathtt{inttree}(k);\mathtt{inttree}(k);\mathtt{int}\},k) \to \mathtt{inttree}(k) \\ \mathtt{siftdown} \mapsto (\{\mathtt{inttree}(k_1);\mathtt{inttree}(k_2);\mathtt{int}(k_3)\},k_4) \to \mathtt{inttree}(k_5),k_6 \\ \mathtt{heapify} \mapsto (\mathtt{intlist}(k_1),k_2) \to \mathtt{inttree}(k_3) \otimes \mathtt{intlist}(k_4),k_5 \\ \mathtt{length} \mapsto (\mathtt{intlist}(k_1),k_2) \to \mathtt{int},k_3 \\ \mathtt{fromList} \mapsto (\mathtt{intlist}(k_1),k_2) \to \mathtt{inttree}(k_3),k_4 \\ \mathtt{leftrem} \mapsto (\mathtt{inttree}(k_1),k_2) \to \mathtt{int} \otimes \mathtt{inttree}(k_3),k_4 \\ \mathtt{delmin} \mapsto (\mathtt{inttree}(k_1),k_2) \to \mathtt{inttree}(k_3),k_4 \\ \mathtt{toList} \mapsto (\mathtt{inttree}(k_1),\mathtt{intlist}(k_2),k_3) \to \mathtt{inttree}(k_4),k_5 \\ \mathtt{sort} \mapsto (\mathtt{intlist}(k_1),k_2) \to \mathtt{intlist}(k_3),k_4 \end{array}\right]$$

Note that the depth of a list is exactly its total size. Thus the overall bound is not likely to improve, but we do gain more precise bounds for the tree manipulations involved. The analyses produces the following results:

```
siftdown: ({t1:inttree(7);t2:inttree(7);w:int},7) -> inttree(7), 0
siftdown: ({t1:inttree(7);t2:inttree(7);w:int},7) -> inttree(7), 0
heapify: (i:int,l:intlist(7),7) -> inttree(7)*intlist(7), 7
length: l:intlist(1) -> int, 0
fromList: (vs:intlist(7),11) -> inttree(7), 11
leftrem: (t:inttree(7),0) -> int*inttree(7), 0
delmin: (t:inttree(7),0) -> inttree(7), 0
```

```
toList: (t:inttree(7),a:intlist(0),13) -> intlist(0), 13
sort: (l:intlist(7),13) -> intlist(0), 13
```

We can see that the overall bound for `sort` is unchanged from the previous analyses, but that where trees are involved we only need space proportional to their depth. In the heap sort, the tree depth will be the logarithm of the supplied list's size. However, this makes these results problematic for use in larger programs — the potential assigned to the trees is logarithmic with respect to the size of the both argument and result lists. Thus not enough potential is assigned to the trees to allow us to assign any potential to the result (the zero in the signature of `sort`). We can reuse the potential on the original unsorted list, though.

The impact of this on the overall bounds of a larger program is that where a program uses the sorted list, and requires stack space proportional to it, the analysis will fail. However, we can still use the original list in such a way and obtain a bound.

Appendix B

Implementation Benchmarks

Below we present a table showing the performance of the implementations on various examples. Note that they were not designed with efficiency in mind, but to test and experiment with the type systems. Thus the figures below should only be taken as a rough indication of the performance of the analyses.

The *analysis* column contains

basic for the simple adaption of Chapter 4,

give-back for the analysis from Chapter 5 or

depth for the depth analysis presented in Chapters 6 and 7.

The *constraints* column presents the number of constraints generated for the linear programming stage, the *inference time* column is the time in milliseconds required to perform the type inference and produce the linear program, and the *solver time* is the time in milliseconds taken to solve the linear program.

The tests were performed on a 2.6GHz Pentium 4 PC with 512MB of main memory, using ocamlopt 3.09.3 for the basic and give-back analyses and SML/NJ version 110.67 for the depth analysis. The linear program solver used throughout was lpsolve 5.5.

Recall that Example 2.11 demonstrates the exponential behaviour due to resource polymorphism. Thus the large number of constraints and long execution time is expected. The number in the leftmost column refers to the number of functions involved, see page 32.

The Huffman tree generation example is an adapted version of the one described in (Jost, 2004b).

Example	Analysis	Constraints	Inference time (ms)	Solver time (ms)
notlist (Ex 2.1)	basic	20	2	2
	give-back	24	1	3
	depth	41	2	5
andlists2 (Ex 5.2)	basic	97	5	5
	give-back	117	6	6
	depth	165	4	8
quicksort	basic	101	8	4
	give-back	143	11	5
	depth	239	12	12
Red black tree insertion	basic	619	43	25
	give-back	880	80	38
	depth	948	38	27
Huffman	basic	527	47	44
	give-back	710	76	61
	depth	882	37	49
heapsort	basic	482	36	29
	give-back	662	65	44
	depth	1035	62	92
Ex 2.11 — 6	basic	906	93	38
8	basic	3690	1201	335
10	basic	14826	39173	7323
6	give-back	1156	175	50
8	give-back	4708	2533	506
10	give-back	18916	64123	9694
6	depth	406	14	31
8	depth	1654	162	395
10	depth	6646	3859	7918

Table B.1: Benchmarking results

Bibliography

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. (2007). Cost analysis of Java bytecode. In *Programming Languages and Systems, 16th European Symposium on Programming (ESOP 2007)*, number 4421 in Lecture Notes in Computer Science, pages 157–172. Springer-Verlag.

Amadio, R. M. (2005). Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1–2):29–60. Also LIF Report 16-2004.

Amadio, R. M. and Dal Zilio, S. (2006). Resource control for synchronous cooperative threads. *Theoretical Computer Science*, 358:229–254.

Appel, A. W. (1992). *Compiling with Continuations*. Cambridge University Press.

Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., and Stark, I. (2005). Mobile resource guarantees for smart devices. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, number 3362 in Lecture Notes in Computer Science, pages 1–26. Springer-Verlag.

Aspinall, D., Hofmann, M., and Konečný, M. (2008). A type system with usage aspects. *Journal of Functional Programming*, 18(2):141–178.

Bellantoni, S. and Cook, S. (1992). A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2(2):97–110.

Birkedal, L., Tofte, M., and Vejlstrup, M. (1996). From region inference to von Neumann machines via region representation inference. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 171–183, New York, NY, USA. ACM Press.

Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. (2007). Automated verification of shape, size and bag properties. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 307–320.

Chin, W.-N. and Khoo, S.-C. (2001). Calculating sized types. *Higher Order and Symbolic Computation*, 14(2-3):261–300.

Chin, W.-N., Khoo, S.-C., Qin, S., Popeea, C., and Nguyen, H. H. (2005a). Verifying safety policies with size properties and alias controls. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 186–195, New York, NY, USA. ACM Press.

Chin, W.-N., Khoo, S.-C., and Xu, D. N. (2003). Extending sized type with collection analysis. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 75–84, New York, NY, USA. ACM Press.

Chin, W.-N., Nguyen, H. H., Qin, S., and Rinard, M. (2005b). Memory usage verification for OO programs. In *Static Analysis, 12th International Symposium (SAS 2005)*, number 3672 in Lecture Notes in Computer Science. Springer-Verlag.

Cook, S. A. (1971). Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1):4–18.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. The MIT Press.

Crary, K. and Weirich, S. (2000). Resource bound certification. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 184–198, New York, NY, USA. ACM Press.

Dantzig, G. B. (1963). *Linear Programming and Extensions*. Princeton University Press.

Debray, S., Lòpez-Garcìa, P., Hermenegildo, M., and Lin, N.-W. (1997). Lower bound cost estimation for logic programs. In *ILPS '97: Proceedings of the 1997 International symposium on Logic Programming*, pages 291–305, Cambridge, MA, USA. MIT Press.

Debray, S. K. and Lin, N.-W. (1993). Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875.

Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*, pages 237–247. ACM Press.

Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (2004). Retrospective: The essence of compiling with continuations. *SIGPLAN Notices*, 39(4):502–514. 20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation.

Grobauer, B. (2001). Cost recurrences for DML programs. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 253–264, New York, NY, USA. ACM Press.

Hermenegildo, M. V., Albert, E., López-García, P., and Puebla, G. (2005). Abstraction carrying code and resource-awareness. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*, pages 1–11, New York, NY, USA. ACM Press.

Hofmann, M. (2000a). Programming languages capturing complexity classes. *SIGACT News*, 31(1):31–42.

Hofmann, M. (2000b). A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289.

Hofmann, M. (2002). The strength of non-size increasing computation. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 260–269, New York, NY, USA. ACM Press.

Hofmann, M. (2003). Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85.

Hofmann, M. and Jost, S. (2003). Static prediction of heap space usage for first-order functional programs. In *POPL '03: Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, New Orleans. ACM Press.

Hofmann, M. and Jost, S. (2006). Type-based amortised heap-space analysis (for an object-oriented language). In Sestoft, P., editor, *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer-Verlag.

Hughes, J. and Pareto, L. (1999). Recursion and dynamic data-structures in bounded space: towards embedded ML programming. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming*, pages 70–81. ACM Press.

Hughes, J., Pareto, L., and Sabry, A. (1996). Proving the correctness of reactive systems using sized types. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 410–423, New York, NY, USA. ACM Press.

Jost, S. (2004a). ARTHUR: A resource-aware typesystem for heap-space usage reasoning. `http://www.tcs.informatik.uni-muenchen.de/~jost/publication.html`.

Jost, S. (2004b). lfd_infer: an implementation of a static inference on heap space usage. In *Proceedings of Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE)*.

Jost, S. (2007). Prototype implementation of space analyses. Deliverable D13, The Embounded Project (IST-510255).

Jost, S. (2008). *Amortised Analysis for Functional Programs*. PhD thesis, Ludwig-Maximilians-University. Forthcoming, provisional title.

Jost, S., Loidl, H.-W., and Hammond, K. (2007a). Report on heap-space analysis. Deliverable D11, The Embounded Project (IST-510255).

Jost, S., Loidl, H.-W., and Hammond, K. (2007b). Report on stack-space analysis (revised). Deliverable D05, The Embounded Project (IST-510255).

Kennedy, A. (2007). Compiling with continuations, continued. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional Programming*, pages 177–190, New York, NY, USA. ACM.

Knuth, D. E. (1971). An empirical study of FORTRAN programs. *Software — Practice and Experience*, 1:105–133.

Konečný, M. (2003). Functional in-place update with layered datatype sharing. In *Typed Lambda Calculi and Applications (TLCA): 6th International Conference*, volume 2701 of *Lecture Notes in Computer Science*, pages 195–210. Springer-Verlag.

Liu, Y. A. and Gòmez, G. (2001). Automatic accurate cost-bound analysis for high-level languages. *IEEE Transactions on Computers*, 50(12):1295–1309.

Marion, J.-Y. and Péchoux, R. (2006). Resource analysis by sup-interpretation. In *Functional and Logic Programming*, number 3945 in Lecture Notes in Computer Science, pages 163–176. Springer-Verlag.

Navas, J., Mera, E., López-García, P., and Hermenegildo, M. V. (2007). User-definable resource bounds analysis for logic programs. In *Logic Programming, 23rd International Conference (ICLP 2007)*, number 4670 in Lecture Notes in Computer Science, pages 348–363. Springer-Verlag.

O'Hearn, P. (2003). On bunched typing. *Journal of Functional Programming*, 13(4):747–796.

Pareto, L. (2000). *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, Göteborg. ISBN 91-7197-979-4.

Paulson, L. C. (1996). *ML for the Working Programmer*. Cambridge University Press, 2nd edition.

Plotkin, G. D. (1975). Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159.

Reistad, B. and Gifford, D. K. (1994). Static dependent costs for estimating execution time. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 65–78, New York, NY, USA. ACM Press.

Reynolds, J. C. (1993). The discoveries of continuations. *LISP and Symbolic Computation*, 6(3–4):233–248.

Röjemo, N. and Runciman, C. (1996). Lag, drag, void and use-heap profiling and space-efficient compilation revisited. In *ICFP '96: Proceedings of the first ACM*

SIGPLAN International Conference on Functional programming, pages 34–41, New York, NY, USA. ACM Press.

Rosendahl, M. (1989). Automatic complexity analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional Programming Languages and Computer Architecture*, pages 144–156, New York, NY, USA. ACM.

Runciman, C. and Wakeling, D. (1993). Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–246.

Simões, H. R., Hammond, K., Florido, M., and Vasconcelos, P. (2007). Using intersection types for cost-analysis of higher-order polymorphic functional programs. In *Types for Proofs and Programs (TYPES 2006)*, number 4502 in Lecture Notes in Computer Science. Springer-Verlag.

Steele, Jr., G. L. (1978). RABBIT: A compiler for SCHEME. Technical Report AITR-474, MIT Artificial Intelligence Laboratory.

Tarjan, R. E. (1985). Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318.

Tofte, M., Birkedal, L., Elsman, M., and Hallenberg, N. (2004). A retrospective on region-based memory management. *Higher Order Symbol. Comput.*, 17(3):245–265.

Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T. H., and Sestoft, P. (2006). Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark.

Tofte, M. and Talpin, J.-P. (1997). Region-based memory management. *Information and Computation*, 132(2).

Unnikrishnan, L., Stoller, S. D., and Liu, Y. A. (2000). Automatic accurate stack space and heap space analysis for high-level languages. Technical Report 538, Computer Science Department, Indiana University.

van Eekelen, M., Shkaravska, O., van Kesteren, R., Jacobs, B., Poll, E., and Smetsers, S. (2007). AHA: Amortized heap space usage analysis. In *Selected Papers of the Eighth Symposium on Trends in Functional Programming (TFP 2007)*. Intellect. To appear.

Vasconcelos, P. (2008). *Size and Cost Analysis using Types*. PhD thesis, University of St Andrews. Forthcoming.

Vasconcelos, P. B. and Hammond, K. (2004). Inferring cost equations for recursive, polymorphic and higher-order functional programs. In Trinder, P., Michaelson, G., and Peña, R., editors, *Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag.

Xi, H. (1998). *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University.