Chapter 17

Prediction of linear memory usage for first-order functional programs

Brian Campbell[1]
Category: Research Paper

Abstract: Hofmann and Jost have presented a type inference system for a pure first-order functional language which uses linear programming to give linear upper bounds on heap memory usage with respect to the input size. We present an extended analysis which infers bounds for heap and stack space requirements, and which uses more expressive post-evaluation bounds to model a common pattern of stack space use.

17.1 INTRODUCTION

Knowing strict upper bounds on the memory usage of a program can guarantee that it will operate without exhausting the available resources. This is particularly important in highly constrained environments, such as smart cards, especially where failures are difficult or expensive to recover from. Hofmann and Jost have presented a type-based approach [6] which can automatically infer such upper bounds for the heap memory usage of a wide range of programs where that usage is linear in the size of the input.

However, their analysis does not include stack memory, so some forms of excessive memory consumption may go unnoticed. In this paper we extend their system to infer linear bounds on the stack, heap or total memory used by programs, and incorporate tail call optimisation in a flexible manner. Of particular

[1]Laboratory for Foundations of Computer Science, University of Edinburgh, `Brian.Campbell@ed.ac.uk`. This research was partially supported by the MRG project (IST-2001-33149).

$$
\begin{array}{l}
\mathrm{P} := \mathsf{let}\ \mathrm{B} \mid \mathsf{let}\ \mathrm{B}\ \mathrm{P} \\
\mathrm{B} := \mathrm{D} \mid \mathrm{D}\ \mathsf{and}\ \mathrm{B} \\
\mathrm{D} := f(x_1,\ldots,x_p) = e_f \\
e := * \mid \mathsf{true} \mid \mathsf{false} \mid x \mid f(x_1,\ldots,x_p) \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \\
\quad \mid (x_1,x_2) \mid \mathsf{match}\ x\ \mathsf{with}\ (x_1,x_2) \to e \\
\quad \mid \mathsf{inl}(x) \mid \mathsf{inr}(x) \mid \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r \\
\quad \mid \mathsf{nil} \mid \mathsf{cons}(x_h,x_t) \quad \mid \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil} \to e_n \mid \mathsf{cons}(x_h,x_t) \to e_c \\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{match}'\ x\ \mathsf{with}\ \mathsf{nil} \to e_n \mid \mathsf{cons}(x_h,x_t) \to e_c
\end{array}
$$

FIGURE 17.1. Syntax

interest is the temporary nature of stack memory usage, which would cause difficulties with a straightforward adaption of the heap analysis. We have overcome this by allowing more precise estimates of free memory *after* an expression has been executed by relating it to both the size of the result *and* the size of variables in the context. This requires some care when part of the result appears in the context.

Finally, we show the soundness of the new analysis with respect to the operational semantics of the language.

17.2 THE LANGUAGE

The language is a simple first-order call-by-value functional programming language. For consistency, it is the one considered by Hofmann and Jost. The syntax for programs is given in Figure 17.1, where f is a function name, $*$ is the value of unit type, 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.

Note the use of variables rather than subexpressions in many places; this corresponds to requiring that the program is in a 'let-normal' form, similar to A-normal form [5], to make the evaluation order explicit and allow the typing rules to be simpler. However, partial compilation of programs of interest is required to put them into that form before we can obtain bounds. Indeed, Jost's implementation of the heap space inference uses an intermediate language produced from a high level language as part of the Mobile Resource Guarantees project [10].

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 deallocation could safely occur. In this language only list elements are heap allocated and we distinguish between (potentially) destructive match expressions and benign, 'read-only', match′ ones. There is more than one possible implementation of heap

management using these marks, but we do not need to pick one. The only subtlety is that it must prevent memory fragmentation from inflating memory usage. See Hofmann and Jost's paper [6] for more discussion on this point.

We presume the existence of some external analysis which ensures that match is used safely so that no data is deallocated while live references to it exist, a property called *benign sharing*. Aspinall and Hofmann's usage aspects [1] or Konečný's DEEL typing [11] are suitable systems. They also provide a conservative estimate of the set of variables that do not overlap on the heap with the result of a given expression in the program. We will make use of this separation property in Section 17.4.

The types are $T := 1 \mid \mathsf{bool} \mid T \otimes T \mid T + T \mid \mathsf{L}(T)$ for unit, boolean, pairs, sums and lists, respectively. The analysis will annotate the types to indicate resource requirements. The language can be extended with richer first-order types such as integers and arbitrary algebraic datatypes without affecting the results. Function signatures are of the form $T_1, \ldots, T_p \to T$.

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 match' so that the argument, `l`, is left intact. As a result, the function needs extra heap memory equal to the space occupied by the argument. Using the 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.

17.2.1 Operational Semantics

Values v in the operational semantics consist of unit, booleans, pairs, variants ($\mathsf{inl}(v)$ and $\mathsf{inr}(v)$ for value v) and heap locations $l \in \mathsf{loc}$ for lists. An environment S maps variables to values, and a store σ maps locations to (value, location) pairs for each list element. A special location null is presumed which can represent nil. The operational semantics is given in Figures 17.2 and 17.3, with judgements of the form

$$m, S, \sigma \vdash^{f,t} e \rightsquigarrow v, \sigma', m'$$

meaning that with m units of free memory, the environment S and the store σ, the expression e (from function f, in tail position if t is true) can be evaluated to value v, with the new store σ' and m' units of free memory. The evaluation of a whole program is realized by the evaluation of a chosen 'initial' function $f(x_1, \ldots, x_p)$ on some arguments $v_1, \ldots, v_p$,

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

where e_f is the body of f.

The operational semantics uses two auxiliary functions to define the memory requirements. The first, size(v), gives the amount of heap memory required to store a value v. In the present setting only list elements are heap allocated, so v is always the pair of a list element's contents and the location of the next element. Secondly, stack(f,g,t) gives the size of stack frame required to call function f from g, in tail position iff t is true. (We assume a fixed frame size for the evaluation of each function, although the system could be adapted to change the frame size when evaluating expressions involving binding.)

Definitions of these functions depend on the concrete implementation, and precise values could be obtained from the compiler. In the examples here we will take the 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 by fixing stack(f,g,t) to be zero everywhere, and stack space by fixing size(v) to be zero.

Some common choices for stack(f,g,t) are to assign every function f a single frame size, frame(f), and set

$$\text{stack}(f,g,t) = \text{frame}(f) \quad \text{for all } g,t,$$

to model a compiler with no tail call optimisation, and

$$\text{stack}(f,g,t) = \begin{cases} \text{frame}(f) & \text{if } t = \mathsf{false} \\ \text{frame}(f) - \text{frame}(g) & \text{otherwise} \end{cases}$$

for general tail call optimisation.

Thus tail-calls are modelled by a combination of the tail position flags and the stack function. Thanks to the use of let-normal form, only the E-LET rule and E-FUN rule have premises with different flags to their conclusion because only the E-LET rule can introduce a subexpression that is not in tail position.

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

We also need to formalise the guarantees that we expect a benign sharing analysis to give. For E-MATCHCONS the 'dead' location should not be accessible from the 'live' variables that the subexpression may use,

$$l \notin \mathscr{R}(\sigma, S[h \mapsto v_h][t \mapsto v_t] \restriction \text{FV}(e_2)),$$

and for E-LET the parts of the heap needed for e_2 should not be altered by match expressions in e_1,

$$\sigma \restriction \mathscr{R}(\sigma, S \restriction \text{FV}(e_2)) = \sigma_0 \restriction \mathscr{R}(\sigma, S \restriction \text{FV}(e_2)),$$

where $\mathscr{R}(\sigma,S)$ is the set of locations in σ reachable from S.

The separation property can also be formalised. For any expression e in the program, the benign sharing analysis should provide a set of variables $V_e \subseteq \text{FV}(e)$

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

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

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

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

$$\frac{S(x) = \mathsf{true} \qquad m,S,\sigma \vdash^{g,t} e_t \rightsquigarrow v,\sigma',m'}{m,S,\sigma \vdash^{g,t} \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \rightsquigarrow v,\sigma',m'}\ \text{(E-IFTRUE)}$$

$$\frac{S(x) = \mathsf{false} \qquad m,S,\sigma \vdash^{g,t} e_f \rightsquigarrow v,\sigma',m'}{m,S,\sigma \vdash^{g,t} \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \rightsquigarrow v,\sigma',m'}\ \text{(E-IFFALSE)}$$

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

$$\frac{S(x) = (v_1,v_2) \qquad m,S[x_1 \mapsto v_1][x_2 \mapsto v_2],\sigma \vdash^{g,t} e \rightsquigarrow v,\sigma',m'}{m,S,\sigma \vdash^{g,t} \mathsf{match}\ x\ \mathsf{with}\ (x_1,x_2) \rightarrow e \rightsquigarrow v,\sigma',m'}\ \text{(E-PAIRELIM)}$$

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

FIGURE 17.2. Operational semantics

that do not overlap with the result of e. More precisely, given an evaluation

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

in the program we have $\mathscr{R}(\sigma,S \restriction V_e) \cap \mathscr{R}(\sigma',v) = \emptyset$. These properties may be derived from (for example) the correctness theorem of [11].

17.3 OVERVIEW

We now present an overview of the Hofmann-Jost heap space analysis and our extensions to bound the stack space requirements.

The Hofmann-Jost system assigns hypothetical amounts of free memory to data structures in proportion to their sizes. Conditions are imposed on the assignments so that the total amount at any point in an evaluation would be sufficient for all allocations and assignments of these amounts to newly constructed data. In particular, the amount assigned to the initial arguments will be an upper bound of the total heap memory requirements of the entire program.

$$\frac{S(x) = \mathsf{inl}(v_0) \qquad m, S[x_l \mapsto v_0], \sigma \vdash^{g,t} e_l \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash^{g,t} \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r \rightsquigarrow v, \sigma', m'} \text{(E-MatchInl)}$$

$$\frac{S(x) = \mathsf{inr}(v_0) \qquad m, S[x_r \mapsto v_0], \sigma \vdash^{g,t} e_r \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash^{g,t} \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r \rightsquigarrow v, \sigma', m'} \text{(E-MatchInr)}$$

$$\frac{}{m, S, \sigma \vdash \mathsf{nil} \rightsquigarrow \mathsf{null}, \sigma, m} \text{(E-Nil)}$$

$$\frac{v = (S(h), S(t)) \qquad l \notin \mathrm{dom}(\sigma)}{m + \mathrm{size}(v), S, \sigma \vdash^{g,t} \mathsf{cons}(h,t) \rightsquigarrow l, \sigma[l \mapsto v], m} \text{(E-Cons)}$$

$$\frac{S(x) = \mathsf{null} \qquad m, S, \sigma \vdash^{g,t} e_1 \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash^{g,t} \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil} \to e_1 \mid \mathsf{cons}(h,t) \to e_2 \rightsquigarrow v, \sigma', m'} \text{(E-MatchNil)}$$

$$\frac{S(x) = l \qquad \sigma(l) = (v_h, v_t) \qquad m + \mathrm{size}((v_h, v_t)), S[h \mapsto v_h][t \mapsto v_t], \sigma \backslash l \vdash^{g,t} e_2 \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash^{g,t} \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil} \to e_1 \mid \mathsf{cons}(h,t) \to e_2 \rightsquigarrow v, \sigma', m'} \text{(E-MatchCons)}$$

$$\frac{S(x) = \mathsf{null} \qquad m, S, \sigma \vdash^{g,t} e_1 \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash^{g,t} \mathsf{match}'\ x\ \mathsf{with}\ \mathsf{nil} \to e_1 \mid \mathsf{cons}(h,t) \to e_2 \rightsquigarrow v, \sigma', m'} \text{(E-Match'Nil)}$$

$$\frac{S(x) = l \qquad \sigma(l) = (v_h, v_t) \qquad m, S[h \mapsto v_h][t \mapsto v_t], \sigma \vdash^{g,t} e_2 \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash^{g,t} \mathsf{match}'\ x\ \mathsf{with}\ \mathsf{nil} \to e_1 \mid \mathsf{cons}(h,t) \to e_2 \rightsquigarrow v, \sigma', m'} \text{(E-Match'Cons)}$$

FIGURE 17.3. Operational semantics continued

This is an amortized analysis, using the *physicist's view* of amortization described by Tarjan [12]. Following Tarjan, we call these hypothetical amounts of free memory *potential*. The Hofmann-Jost analysis assigns potential to data structures using type annotations, and side conditions on the typing rules ensure that the assignments are sufficient for any allocations.

The Hofmann-Jost system annotates typings and function signatures with non-negative rational values[2] in two places. First, we add 'before' and 'after' amounts to typing judgements and function signatures to represent constant amounts of potential. The constraints on these will mirror the operational semantics, requiring the 'before' annotation at an allocation to be at least as large as the amount to be allocated, and the 'after' annotation to be correspondingly lower. Similarly, a relative increase in the annotations is permitted when typing a deallocation.

Second, we place annotations on list types to denote 'per-element' amounts of potential. So a list x of type $\mathsf{L}(T,k)$ represents a free memory requirement of $k \times |x|$ units, and at a $\mathsf{cons}(h,x)$ expression we require that the 'after' annotation

[2]Fractional annotations can arise naturally in this system. For example, if we require a cell for every second element of a boolean list l, then l will have the type $\mathsf{L}(\mathsf{bool}, \frac{1}{2})$.

must be *k extra* units lower to compensate for the longer list. Then wherever an element is taken from a list with that type using match, the constraints in the typing allow k more units in the constant memory annotation. More intuitively, the cons reserves k units of free memory per element, and the match releases it again for use.

In this way we can represent affine memory requirements. For example, the `notlist` function might be given the function signature

$$\texttt{notlist} : \mathsf{L}(\mathsf{bool},3),0 \to \mathsf{L}(\mathsf{bool},2),0,$$

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 length list can transform a 'constant' amount of potential into a 'per-element' one. For example, in

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

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

To infer these annotations the typing rules give linear constraints that their values must satisfy. We can use standard linear programming techniques to solve these constraints to find a minimal set of satisfying annotations.

The straightforward part of adapting the Hofmann-Jost analysis to include stack space in the bounds is to adjust the constraints to require extra potential at each function call for the stack space required. To model tail call optimisation we reproduce the tail position information from the operational semantics in the typing rules. Note that it would be sound to use a more conservative stack(f,g,t) function in the type system than a particular implementation requires. This allows the analysis to provide coarser upper bounds which covers multiple implementations of the language.

However, the temporary nature of stack memory usage is not handled well by the straightforward adaptions above. A simple example involves 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
```

where one stack frame per element (i.e. $|\texttt{l}|$ frames) is required. This stack memory is free again after the function returns, but only a constant amount of potential can be assigned to the result because there is no annotation on the result's type that is capable of representing $|\texttt{l}|$ frames. Hence, 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 analysis sums the requirements of the two `length l` calls. So the best stack memory bound on `twicelength` is twice the actual usage. While this example is contrived, reuse of variables (and the corresponding requirement for stack space) occurs frequently, such as when consulting a lookup table repeatedly.

For a more subtle example consider 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)
```

which computes the pairwise boolean 'and' of two lists. The size of either list would be an appropriate upper bound on the stack space required, because the actual amount used is the size of the shorter list. A straightforward adaption of Hofmann-Jost as outlined above can infer this without difficulty. (In these examples we estimate only the stack memory because the problem affects stack space analysis more acutely, so let size $\equiv 0$ and $\mathrm{stack}(f,g,t) = 1$ for all f,g,t.)

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. We do not handle maxima in our analysis (this is considerably more involved, and the topic of future work), but we would still expect to be able to infer a bound of $|\texttt{l1}|$ frames. However, a straightforward adaption would again simply sum the requirements because the post-evaluation potential for the first call can only be expressed in terms of the result, `r1`. Ideally, we ought to be able to reuse the potential initially assigned to `l1` at the second function call.

We achieve this reuse by adding a second 'give-back' annotation to list types which indicates an amount to be passed to the 'normal' annotation of a future use of the variable. The above functions will then have types whose constraints allow the following solutions:

$$\begin{aligned}
\texttt{length} &: \mathsf{L}(T, 1 \rightsquigarrow 1), 0 \to \mathsf{int}, 0 \\
\texttt{twicelength} &: \mathsf{L}(T, 1 \rightsquigarrow 1), 1 \to \mathsf{int}, 1 \\
\texttt{andlists} &: \mathsf{L}(\mathsf{bool}, 1 \rightsquigarrow 1), \mathsf{L}(\mathsf{bool}, 0 \rightsquigarrow 0), 0 \to \mathsf{L}(\mathsf{bool}, 0 \rightsquigarrow 0), 0 \\
\texttt{andlists2} &: \mathsf{L}(\mathsf{bool}, 1 \rightsquigarrow 1), \mathsf{L}(\mathsf{bool}, 0 \rightsquigarrow 0), \mathsf{L}(\mathsf{bool}, 0 \rightsquigarrow 0), 1 \\
&\qquad \to \mathsf{L}(\mathsf{bool}, 0 \rightsquigarrow 0) \otimes \mathsf{L}(\mathsf{bool}, 0 \rightsquigarrow 0), 1
\end{aligned}$$

These signatures mean that if any of these functions is given a list l as the first argument, then $1 \times |l|$ stack frames are sufficient for evaluation, and that those $1 \times |l|$ stack frames will be free afterwards[3]. As the free memory afterwards is

[3] Plus one more for `twicelength` and `andlists2`, due to the extra function call.

expressed as an annotation on l, its reuse will be taken into account when typing a subsequent occurrence of l.

To use these extra annotations, we provide an alternative to the Hofmann-Jost rule for contraction which takes advantage of the 'give-back' annotation. We choose to adapt the LET rule to ensure that the uses of the variables involved (Δ) occur sequentially. We also change the rules for match so that their constraints not only 'release' the potential for each list element, but also 'reserve' the give-back potential.

17.4 THE TYPE SYSTEM

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

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

where k_l, k_r, k and k' are constraint variables. Sum types are also annotated to reflect different resource requirements depending upon the choice made, but are not assigned a give-back annotation because the separation condition we use to ensure soundness requires the values involved to be heap allocated. To link the annotations to amounts of memory, we define a function which sums the annotations over every reachable value:

$$\begin{aligned}
\Upsilon &: \mathsf{heap} \times \mathsf{val} \times T_a \to \mathbb{Q}^+, \\
\Upsilon(\sigma, *, 1) = \Upsilon(\sigma, c, \mathsf{bool}) &= \Upsilon(\sigma, \mathsf{null}, \mathsf{L}(T, k \rightsquigarrow k')) = 0, \\
\Upsilon(\sigma, (v_1, v_2), T_1 \otimes T_2) &= \Upsilon(\sigma, v_1, T_1) + \Upsilon(\sigma, v_2, T_2), \\
\Upsilon(\sigma, \mathsf{inl}(v), (T_l, k_l) + (T_r, k_r)) &= k_l + \Upsilon(\sigma, v, T_l), \\
\Upsilon(\sigma, \mathsf{inr}(v), (T_l, k_l) + (T_r, k_r)) &= k_r + \Upsilon(\sigma, v, T_r), \\
\Upsilon(\sigma, l, \mathsf{L}(T, k \rightsquigarrow k')) &= \Upsilon(\sigma, \sigma(l), T \otimes \mathsf{L}(T, k \rightsquigarrow k')) + k,
\end{aligned}$$

and extend it to environments:

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

For example, if x is a list of booleans, then $\Upsilon(\sigma, S(x), \mathsf{L}(\mathsf{bool}, k \rightsquigarrow k'))$ is k times the length of x.

Similarly, we define a counterpart to measure the potential for the give-back

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

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

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

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

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

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

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

FIGURE 17.4. Typing rules for expressions

annotations:

$$\begin{array}{rl} & \Upsilon' : \mathsf{heap} \times \mathsf{val} \times T_a \times \mathsf{loc} \to \mathbb{Q}^+ \\ \Upsilon'(\sigma, *, 1, l) = \Upsilon'(\sigma, c, \mathsf{bool}, l) = & \Upsilon'(\sigma, \mathsf{null}, \mathsf{L}(T, k \rightsquigarrow k'), l) = 0 \\ \Upsilon'(\sigma, (v_1, v_2), T_1 \otimes T_2, l) = & \Upsilon'(\sigma, v_1, T_1, l) + \Upsilon'(\sigma, v_2, T_2, l) \\ \Upsilon'(\sigma, \mathsf{inl}(v), (T_l, k_l) + (T_r, k_r), l) = & \Upsilon'(\sigma, v, T_l, l) \\ \Upsilon'(\sigma, \mathsf{inr}(v), (T_l, k_l) + (T_r, k_r), l) = & \Upsilon'(\sigma, v, T_r, l) \\ \Upsilon'(\sigma, l', \mathsf{L}(T, k \rightsquigarrow k'), l) = & \Upsilon'(\sigma, \sigma(l'), T \otimes \mathsf{L}(T, k \rightsquigarrow k'), l) + \begin{cases} k' & \text{if } l' = l \\ 0 & \text{otherwise} \end{cases} \end{array}$$

The amount is measured *per heap location l* so that we can use the heap separation condition in the LET case of the proof.

Throughout, we adopt the convention that for any type $\mathsf{L}(T, k \rightsquigarrow k')$ we add the constraint $k \geq k'$ to prevent unused variables in the context receiving unbounded give-back annotations. This reduces the amount of clutter in the type system.

The typing rules for expressions in the basic system are given in Figures 17.4 and 17.5, and use the size and stack functions from the operational semantics, except that size now operates on types rather than values. This assumes that all values of the same type require the same storage. As before, heap or stack space can be considered on its own by setting one of the functions to zero everywhere.

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

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

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

$$\frac{}{\Gamma, n \vdash^{g,t}_{\Sigma} \mathsf{nil} : \mathsf{L}(T, k), n' \mid \{n \geq n'\}} \quad \text{(NIL)}$$

$$\frac{}{\Gamma, h : T, t : \mathsf{L}(T, k), n \vdash^{g,t}_{\Sigma} \mathsf{cons}(h, t) : \mathsf{L}(T, k), n' \mid \{n \geq k + n' + \mathrm{size}(T \otimes \mathsf{L}(T, k))\}} \quad \text{(CONS)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash^{g,t}_{\Sigma} e_n : T', n' \mid \Phi_n \qquad \Gamma, h : T, t : \mathsf{L}(T, k \rightsquigarrow k'), n_c \vdash^{g,t}_{\Sigma} e_c : T', n'_c \mid \Phi_c \\ \Phi = \Phi_n \cup \Phi_c \cup \{n_c = n + k + \mathrm{size}(T \otimes \mathsf{L}(T, k)), n'_c = n' + k'\}\end{array}}{\Gamma, x : \mathsf{L}(T, k \rightsquigarrow k'), n \vdash^{g,t}_{\Sigma} \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil} \to e_n \mid \mathsf{cons}(h, t) \to e_c : T', n' \mid \Phi} \quad \text{(LISTELIM)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash^{g,t}_{\Sigma} e_n : T', n' \mid \Phi_n \qquad \Gamma, h : T, t : \mathsf{L}(T, k \rightsquigarrow k'), n_c \vdash^{g,t}_{\Sigma} e_c : T', n'_c \mid \Phi_c \\ \Phi = \Phi_n \cup \Phi_c \cup \{n_c = n + k, n'_c = n' + k'\}\end{array}}{\Gamma, x : \mathsf{L}(T, k \rightsquigarrow k'), n \vdash^{g,t}_{\Sigma} \mathsf{match}'\ x\ \mathsf{with}\ \mathsf{nil} \to e_n \mid \mathsf{cons}(h, t) \to e_c : T', n' \mid \Phi} \quad \text{(LISTELIM}')$$

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

FIGURE 17.5. Typing rules for expressions (continued)

Typing judgements for expressions take the form

$$\Gamma, n \vdash^{g,t}_{\Sigma} e : T, n' \mid \Phi$$

where Γ is the typing context, n is an amount of potential before evaluation (in addition to that from the type annotations), n' is the corresponding amount after evaluation, T is the annotated type of e, Σ contains the function signatures, g is the function in which the expression appears, t is the tail position flag and Φ is the set of constraints on annotations that must hold for a valid typing.

The function signatures now take the form $T_1, \ldots, T_p, k \to T, k'$ where T_i and T are the annotated types for the arguments and the result and k and k' are extra amounts of potential required and released (analogous to n and n' above).

The typing rules for function definitions are given in Figure 17.6. They check that mutually recursive blocks of functions and entire programs are well typed, with functions conforming to their function signatures in Σ. A program P is well-typed if we can derive $\vdash_{\Sigma} P \Rightarrow \Phi$ for some Φ.

There are two rules for contraction. The SHARE rule is equivalent to its coun-

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

$$\frac{\vdash_\Sigma \mathrm{D} \Rightarrow \Phi \qquad \vdash_\Sigma \mathrm{B} \Rightarrow \Phi'}{\vdash_\Sigma \mathrm{D}\ \mathsf{and}\ \mathrm{B} \Rightarrow \Phi \cup \Phi'} \qquad \frac{\vdash_\Sigma \mathrm{B} \Rightarrow \Phi}{\vdash_\Sigma \mathsf{let}\ \mathrm{B} \Rightarrow \Phi} \qquad \frac{\vdash_\Sigma \mathrm{B} \Rightarrow \Phi \qquad \vdash_\Sigma \mathrm{P} \Rightarrow \Phi'}{\vdash_\Sigma \mathsf{let}\ \mathrm{B}\ \mathrm{P} \Rightarrow \Phi \cup \Phi'}$$

FIGURE 17.6. Typing rules for function definitions

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

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

$$\frac{T = T_1 \oplus T_2 \mid \Phi \qquad T' = T_1' \oplus T_2' \mid \Phi' \qquad \Phi_0 = \Phi \cup \Phi' \cup \{k = k_1 + k_2, k' = k_1' + k_2'\}}{(T,k) + (T',k') = (T_1,k_1) + (T_1',k_1') \oplus (T_2,k_2) + (T_2',k_2') \mid \Phi_0}$$

$$\frac{T = T_1 \oplus T_2 \mid \Phi}{\mathsf{L}(T,k) = \mathsf{L}(T_1,k_1) \oplus \mathsf{L}(T_2,k_2) \mid \Phi \cup \{k = k_1 + k_2\}}$$

FIGURE 17.7. Rules for splitting annotations

terpart in Hofmann-Jost. It splits the potential between two uses of the variable. Informally, we require this splitting because the total bound on the free memory required with respect to some variable x is the sum of the bounds with respect to each individual use of x. The auxiliary rules in Figure 17.7 formalise this splitting, which ensures that the types' annotations sum pairwise to the combined type. For example, the judgement

$$\mathsf{L}(\mathsf{bool},k) = \mathsf{L}(\mathsf{bool},k_1) \oplus \mathsf{L}(\mathsf{bool},k_2) \mid \{k = k_1 + k_2\}$$

allows $\mathsf{L}(\mathsf{bool},3) = \mathsf{L}(\mathsf{bool},2) \oplus \mathsf{L}(\mathsf{bool},1)$, 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.

The second (new) form of contraction is part of the LET rule. This allows the potential given-back in e_1 to be used in e_2. The 4-place relation $\cdot = \cdot \succ \cdot \mid \cdot$ given in Figure 17.8 formalises the use of the give-back annotations. For example, the judgement

$$x : \mathsf{L}(\mathsf{bool},k \rightsquigarrow k') = x : \mathsf{L}(\mathsf{bool},k_1 \rightsquigarrow k_1') \succ x : \mathsf{L}(\mathsf{bool},k_2 \rightsquigarrow k_2') \mid \{k \geq k_1, k - k_1 + k_1' \geq k_2, k_2' \geq k'\}$$

means that at the first use of x, k_1' out of the k_1 annotation is only needed temporarily, so it can be reused at the second occurrence of x (as part of k_2).

The separation condition for Δ and the result of e_1 in LET is required to ensure that no give-back potential is assigned to both e_1's result and Δ_1. For example, the

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

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

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

$$\frac{T = T_1 \succ T_2 \mid \Phi \qquad \Phi' = \{k \geq k_1, k - k_1 + k_1' \geq k_2, k_2' \geq k'\}}{\mathsf{L}(T, k \rightsquigarrow k') = \mathsf{L}(T_1, k \rightsquigarrow k_1') \succ \mathsf{L}(T_2, k_2 \rightsquigarrow k') \mid \Phi \cup \Phi'}$$

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

FIGURE 17.8. Rules for contraction in let expressions

identity function `let identity l = l` can be given the function signature

$$\texttt{identity} : \mathsf{L}(\mathsf{bool}, 1 \rightsquigarrow 1), 0 \to \mathsf{L}(\mathsf{bool}, 1 \rightsquigarrow 1), 0.$$

If we did not require separation we could use the potential assigned to `l` twice in an expression `let x = identity l in ... x ... l ...`, once for `x` and once for `l` (via the 'give-back' annotation), and the space bound would be too low.

We presumed in Section 17.2 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 [11] can be used. Thus we can use this set during type inference to decide which variables from the context to put into Δ. Then we construct Δ_1 and Δ_2 from Δ using fresh constraint variables and derive the set of constraints Φ_3 from the rules for $\succ$.

We also need to adapt other rules so that the give-back annotations reflect free memory at runtime which is not accounted for by other annotations. The LISTELIM′ rule adds the 'normal' list annotation to the constant annotation when typing the cons subexpression because we are extracting an element from x. Our replacement rule also requires the amount for the 'give-back' annotation to be returned afterwards for a later use of x: We also adapt LISTELIM in the same way.

17.4.1 Soundness

The soundness 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 annotations also conservatively predict the free memory afterwards. Moreover, execution will not consume any extra free memory (q) that may be available.

Note that $\Upsilon(\sigma,S,\Gamma)$ is defined when the variables in Γ have corresponding values in S and σ that are of the correct type. The $\max\{0,\dots\}$ is present in the 'after' bound to prevent newly allocated data structures interfering with the potential calculation.

Theorem 17.1. *If an expression e in a well-typed program has a typing*

$$\Gamma, n \vdash^{g,t}_{\Sigma} e : T, n' \mid \Phi$$

with an assignment of nonnegative rationals to constraint variables which satisfies the constraints generated for the whole program, 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

$$m, S, \sigma \vdash^{g,t} e \rightsquigarrow v, \sigma', m'$$

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

Proof. (Sketch.) 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.

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

$$m \geq n + \Upsilon(\sigma,S,\Gamma) + q = n + \Upsilon(\sigma, S_0[a \mapsto v_x, b \mapsto v_x], (\Gamma_0, a : T_1, b : T_2)) + q$$

by the linearity of Υ with respect to $\oplus$. 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 linearity of Υ'.

LET. Using the definition of Υ, Υ' and $\succ$ it can be shown that

$$\begin{aligned} \Upsilon(\sigma,S,\Delta) &\geq \Upsilon(\sigma,S,\Delta_1)), \\ \Upsilon(\sigma,S,\Delta) - \Upsilon(\sigma,S,\Delta_1) + \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma,S,\Delta_1,l) &\geq \Upsilon(\sigma,S,\Delta_2) \quad \text{and} \\ \Upsilon'(\sigma,S,\Delta_2,l) &\geq \Upsilon'(\sigma,S,\Delta,l). \end{aligned}$$

The first is used to apply the induction hypothesis to e_1. The remainder $\Upsilon(\sigma,S,\Delta) - \Upsilon(\sigma,S,\Delta_1)$ is kept aside for e_2 by incorporating it into the constant q.

To form the precondition for applying the induction hypothesis to e_2 we note that $\Upsilon'(\sigma_0,v_0,T_0,l) = 0$ for all $l \in \mathscr{R}(\sigma, S \restriction \mathrm{dom}(\Delta_1))$ can be deduced from the separation condition $\mathscr{R}(\sigma, S \restriction \mathrm{dom}(\Delta)) \cap \mathscr{R}(\sigma_0,v_0) = \emptyset$. So

$$\sum_{l \in \mathsf{loc}} \max\{0, \Upsilon'(\sigma,S,\Delta_1,l) - \Upsilon'(\sigma_0,v_0,T_0,l)\} = \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma,S,\Delta_1,l),$$

after which the second property using the amount left aside is sufficient. Then using the third property the result of the induction hypothesis for e_2 can be transformed to show the result for the let.

The other cases take two forms. Those with a subexpression (IF, PAIRELIM, SUMELIM, LISTELIM, LISTELIM′) adjust the context as necessary, extracting part of the amount from an annotation if matching something, then invoke the induction hypothesis. The others (UNIT, BOOL, VAR, PAIR, INL, INR, NIL, CONS) use $\Upsilon(\sigma, v, A) \geq \sum_{l \in \mathsf{loc}} \Upsilon'(\sigma, v, T, l)$ to deal with unused parts of the context and then follow from simple arithmetic manipulations. Function application (FUN) mixes the two forms. □

We can extend the result to the whole program:

Corollary 17.2. *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*

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

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

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

units of memory, for any assignment of nonnegative rationals to constraint variables which satisfies the generated constraints.

The soundness result can be extended to non-terminating programs, by adjusting the operational semantics to allow the execution to be terminated at any point. The proof then shows that any such partial execution also respects the inferred memory bound. Moreover, if the tail call optimisation extension is not used, then obtaining a stack space bound also gives us a bound on the call depth, guaranteeing termination. The 'resource polymorphism' extension suggested by Hofmann and Jost which allows functions to have different signatures at different points in the program can also be applied. A full proof of the soundness theorem for the analysis with these extensions and a slightly larger language can be found in [2].

An implementation of the give-back system with tail-recursion analysis has been produced[4], based upon Jost's `lfd_infer` [10]. This system has only a small linear increase in the number of constraints and variables in the linear programs used, and so should enjoy similar efficiency to Jost's system.

17.5 RELATED AND FURTHER WORK

There are other examples of the use of type systems to provide guarantees on memory usage. Pareto and Hughes [8] gave a system based on *sized-types* which can be used to check heap and stack memory bounds expressed in Presburger arithmetic, using explicit regions to handle deallocation. Crary and Weirich [4]

[4] Available from `http://homepages.inf.ed.ac.uk/bcampbe2/tfp08/`.

present a flexible type system to certify execution time. Both of these system check bounds, but do not infer them, although Chin and Khoo's work have used transitive closure and widening operations to infer sizes in the form of Presburger formulae on the input's size [3].

Jost has extended the basic analysis in [6] to include higher-order functions along with a some other improvements [9], and one potential avenue of further work is to merge that with the enhancements presented in this paper. Hofmann and Jost have also presented a type system (without inference) for checking resource bounds of object-orientated programs [7].

Another area of interest is to introduce annotations which give resource bounds in terms of the *depth* of data structures, rather than their total size. Ongoing work in this area involves representing several alternate annotations in contexts, for example, one for each branch of a tree. This should be especially useful for inferring stack bounds of programs using tree structured data. It may also express bounds that are the maximum of two other bounds, which would be useful for examples like `andlists2` in Section 17.3.

17.6 CONCLUSION

We have shown how to add stack bounds to Hofmann and Jost's analysis, and a mechanism for obtaining better overall bounds by using more expressive post-evaluation calculation of potential. This is particularly important for obtaining good stack space bounds because of its temporary nature.

REFERENCES

[1] D. Aspinall, M. Hofmann, and M. Konečný. A type system with usage aspects. *Journal of Functional Programming*, 18(2):141–178, 2008.

[2] B. Campbell. *Type-based amortized stack memory prediction*. PhD thesis, University of Edinburgh, 2008.

[3] W.-N. Chin and S.-C. Khoo. Calculating sized types. *Higher Order and Symbolic Computation*, 14(2-3):261–300, 2001.

[4] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 184–198, New York, NY, USA, 2000. ACM Press.

[5] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. 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, 1993.

[6] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, 2003. ACM Press.

[7] M. Hofmann and S. Jost. Type-based amortised heap-space analysis (for an object-oriented language). In P. Sestoft, editor, *Proceedings of the 15th European Sympo-*

sium on Programming (ESOP), Programming Languages and Systems, volume 3924 of *LNCS*, pages 22–37. Springer-Verlag, 2006.

[8] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ML programming. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, pages 70–81. ACM Press, 1999.

[9] S. Jost. ARTHUR: A resource-aware typesystem for heap-space usage reasoning. http://www.tcs.informatik.uni-muenchen.de/ jost/publication.html, 2004.

[10] S. Jost. 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)*, 2004.

[11] M. Konečný. 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, 2003.

[12] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.