Static Prediction of Heap Space Usage for First-Order Functional Programs

Martin Hofmann Steffen Jost

LMU München, Institut für Informatik
Oettingenstraße 67, 80538 München, Germany
{mhofmann, jost}@informatik.uni-muenchen.de

ABSTRACT

We show how to efficiently obtain linear a priori bounds on the heap space consumption of first-order functional programs.

The analysis takes space reuse by explicit deallocation into account and also furnishes an upper bound on the heap usage in the presence of garbage collection. It covers a wide variety of examples including, for instance, the familiar sorting algorithms for lists, including quicksort.

The analysis relies on a type system with resource annotations. Linear programming (LP) is used to automatically infer derivations in this enriched type system.

We also show that integral solutions to the linear programs derived correspond to programs that can be evaluated without any operating system support for memory management. The particular integer linear programs arising in this way are shown to be feasibly solvable under mild assumptions.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; D.1.1 [**Programming Techniques**]: Applicative (functional) programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*

General Terms

Languages, Theory, Reliability, Performance.

Keywords

Functional Programming, Resources, Heap, Garbage Collection, Program Analysis.

1. INTRODUCTION

This paper addresses the following problem. Given a functional program containing a function f of type, say, $\mathsf{L}(\mathsf{B}) \to \mathsf{L}(\mathsf{B})$, i.e., turning lists of booleans into lists of booleans find a function v such that the the computation $f(w)$ requires no more than $v(w)$ additional heap cells.

In this generality, the problem admits the following trivial solution: We can instrument the code for f by a counter that is augmented each time we require allocation of a heap cell. The function v is then the function computed by this instrumented code followed by a projection that discards the output and only keeps the value of the counter.

Even if we require that v depend only on the length of the input w and not w itself, we could for a given input length l run the instrumented code on all boolean lists of length l and take the maximum. We still have a computable function that bounds the heap space required by the computation of f.

This trivial solution suffers from two flaws. First, evaluating v requires as many resources as evaluating f itself. Moreover, even though the code for v constitutes a mathematical description of the bounding function v, it is in a form that allows one to say very little about its global behaviour. Both flaws are unacceptable in a scenario where independently verifiable certificates on resource usage of mobile code are desired [14, 1].

What one would rather expect in this situation is a statement of the form: running f on an input of length n will require no more than $b(n)$ heap cells where $b(n)$ is an *expression* like $3n+7$ or $2.5n^3+4n^2$ or $2^{1.5n}$. It is only from such an expression that one can glean immediate information about the expected behavior of the code to be run.

In this paper we describe a method for automatically obtaining *linear* bounds on the heap space usage of functional programs. Of course, it is undecidable whether a given program admits such a linear bound, so we must accept certain restrictions. We claim, however, that the restrictions we make are quite natural and moreover, our analysis is provably efficient in this case.

An important limitation of our work is that only first-order programs are considered. This means that a program is a mutual recursive definition of first-order top level functions. While perhaps being against the credo of functional programming it offers us surprising benefits and moreover many uses of higher-order functions are actually a definitional extension of first-order functional programming: in principle one can eliminate them by code duplication. We

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright 2003 ACM 1-58113-628-5/03/0001 ...$5.00.

comment on this and on the difficulties encountered with fully general higher-order functions later in Section 11.

1.1 Overview of results

We assume an operational semantics that maintains a freelist which is reduced whenever a constructor function like cons is evaluated. On the other hand, we assume that certain pattern matches returns the matched cell to the freelist which accordingly increases in the branches of the match. If we try to evaluate a constructor under an insufficiently large freelist the evaluation gets stuck.

We then devise an annotation of typing derivations with nonnegative rational values which allows for prediction of the freelist size required to evaluate the program. For instance, if we derive $x : \mathsf{L}(\mathsf{L}(\mathsf{B},1),2),3 \vdash e : \mathsf{L}(\mathsf{B},4),5$ then this signifies that if we evaluate e in a situation which binds x to a list $[l_1,\dots,l_m]$ then a freelist of size at least $3+2m+1\sum_i|l_i|$ suffices to prevent evaluation from getting stuck. If the evaluation terminates with a result l then the freelist will have size $5+4|l|$. Here $|\cdot|$ denotes the length of a list.

We note two crucial features: First, the size estimate for the freelist left after evaluation is given as a function of the result type rather than the input. Second, estimates do not just depend on the overall size of arguments but may attach different weight to various parts of the data. In the example the length of the input list counts twice, whereas the lengths of the component lists only count once. We find that these features allow for a surprisingly smooth compositional formulation of the annotations.

Given a concrete program P we then set up a "skeleton" of an annotated derivation which contains variables in place of actual annotations. The various side conditions in our rules then take the form of linear inequalities between these variables. We thus obtain a *linear program* $\mathcal{L}(P)$ whose solutions are in one-to-one correspondence to valid annotations. As is well-known such solutions can be efficiently computed.

We also show that *integral* solutions to the $\mathcal{L}(P)$ are in 1-1 correspondence to enriched versions of P in the programming language LFPL [8] which bypasses memory management by explicitly passing around memory cells as part of the data. Programs in LFPL largely behave like imperative programs that modify heap-allocated data in-place rather than claiming fresh memory for results of computations and returning unused memory. In this way, our inference can also be viewed as type inference for LFPL.

It must be said, though, that not all possible LFPL programs arise as reconstructions from solutions of the constraint system. The problem of reconstructing arbitrary LFPL programs is considered in more detail in [11].

While obtaining integral solutions to linear programs is in general NP hard, we prove that in several important and natural sub-cases of our setting they can be obtained efficiently.

We emphasize that our functional programs are not necessarily required to be linearly typed. Indeed, we have a contraction rule corresponding to aliasing that allows us to identify two variables provided we split the resource annotations accordingly. For example, if we have $x{:}\mathsf{L}(\mathsf{B},3), y{:}\mathsf{L}(\mathsf{B},6), 5 \vdash e : C, 6$ then the contraction rule allows us to derive $z{:}\mathsf{L}(\mathsf{B},9), 5 \vdash e : C, 6$. Operationally, x, y point to a shared memeory region.

If we use this contraction rule then validity of our analysis relies on the following semantic condition: if at any point in the evaluation of a program a heap cell is deallocated in a destructive pattern match then this cell must not be accessible from the variables occurring in the remaining program fragment. We speak of *benign sharing* in this case. A violation of the property is called *malignant sharing*.

Notice that if a program exhibits malignant sharing then it will not necessarily crash due to null pointer access because it might not actually follow the path to the dangling reference even though this is possible. One may thus compare benign sharing to the property ensured by garbage collection.

We formalise benign sharing on the level of the operational semantics as a judgment $S,\sigma \vdash e \rightsquigarrow^{\mathrm{bs}} v,\sigma'$ which asserts that in stack S and heap σ the evaluation of e results in value v and new heap σ' and, moreover, all sharing during that evaluation is benign.

For particular programs we may be able to assert benign sharing by inspection or logical reasoning. More interestingly, we would like to guarantee it by some static type system. We already know that linear typing, i.e., the absence of contraction, provides such a guarantee; we conjecture that the more general read-only type systems and analyses described in [2, 12, 15, 18] all are able to provide such a guarantee as well, by suitably restricting but not altogether excluding the contraction rule.

The important point here is that the semantic formalisation of benign sharing makes no reference to resource annotations so that discharging the extra assumption made is orthogonal to the work described in this paper.

We also mention that, of course, we can recursively define *cloning functions* in the strictly linear fragment, for instance $\mathtt{clone} : \mathsf{L}(\mathsf{B},2) \to \mathsf{L}(\mathsf{B},0) \otimes \mathsf{L}(\mathsf{B},0)$. The two copies returned are not aliased but one of them is constructed using fresh heap space.

Notation: The set of natural numbers denoted $\mathbb{N}$ is assumed to contain zero. We let $\mathbb{Q}^+$ denote the set of nonnegative rational numbers.

If f is a finite function we write $f \setminus x$ for $f\restriction(\operatorname{dom} f \setminus \{x\})$, that is, the restriction of f to its domain less the element x. We write $f[x{\mapsto}v]$ to denote the finite function that maps x to v and acts like f otherwise.

$\mathrm{FV}(e)$ denotes the set of free variables occurring within the term e. The substitution of a free variable v by t in term e is denoted by $e[t/v]$.

If l denotes a list, then $|l|$ denotes the length of the list. Equivalently, $|l|$ is the number of nodes of l in a machine representation.

Acknowledgements: Part of this research was carried out within the EU project IST-2001-33149 "Mobile Resource Guarantees". We also acknowledge financial support by the *Deutsche Forschungsgemeinschaft* (DFG).

2. FUNCTIONAL LANGUAGE

We define a first-order typed functional language LF as follows.

$$\begin{aligned} &\text{zero-order types:} && A ::= \mathsf{1} \mid \mathsf{B} \mid \mathsf{L}(A) \mid A \otimes A \mid A + A \\ &\text{first-order types:} && F ::= (A,\dots,A) \to A \end{aligned}$$

Here B is the type of Booleans, $\mathsf{L}(A)$ is the type of lists with entries from A, sum and product are denoted by $+, \otimes$. Finally, $\mathsf{1}$ is a singleton type. We can also include labelled trees, but refrain from doing so to save space. However, one of our examples uses trees.

Since we are interested in memory consumption, we define at this point a function $\mathsf{SIZE} : \mathsf{LF}$-type $\to \mathbb{N}$ for later use:

$$\begin{aligned}
&\mathsf{SIZE}\,(1) = \mathsf{SIZE}\,(\mathsf{B}) = \mathsf{SIZE}\,(\mathsf{L}(A)) = 1\\
&\mathsf{SIZE}\,(A \otimes C) = \mathsf{SIZE}\,(A) + \mathsf{SIZE}\,(C)\\
&\mathsf{SIZE}\,(A + C) = 1 + \max\bigl(\mathsf{SIZE}\,(A)\,, \mathsf{SIZE}\,(C)\bigr)
\end{aligned}$$

To save space we omit the grammar for LF terms as well as the typing rules. Both can be reconstructed from the typing rules for the annotated version $\mathsf{LF}_\Diamond$, given in Section 4.

All that needs to be done is to remove all reference to resource constraints as shown in the following example of the two variable rules.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_\Sigma x{:}\Gamma(x)}\ (\mathsf{LF}{:}\textsc{Var}) \qquad \frac{x \in \mathrm{dom}(\Gamma) \quad n \geq n'}{\Gamma, n \vdash_\Sigma x{:}\Gamma(x), n'}\ (\mathsf{LF}_\Diamond{:}\textsc{Var})$$

We draw attention to the presence of the two pattern matching constructs match and match' governed by rules $\mathsf{LF}_\Diamond$:List-Elim and $\mathsf{LF}_\Diamond$:List-Elim'. These rules differ only in the resource annotations so that both pattern matches have identical LF-typing rules. The semantics of match is a destructive one: the list node being matched against is deallocated afterwards; in a match' the matched node is preserved for subsequent use.

We also point out that the typing rules for $\mathsf{LF}_\Diamond$, hence the typing rules for LF, are formulated in a linear style. That is, multiple occurrences of a variables are explicitly introduced via the rule $\mathsf{LF}_\Diamond$:Share. For convenience, we give the LF-version here:

$$\frac{\Gamma, x{:}A, y{:}A \vdash_\Sigma e{:}C}{\Gamma, z{:}A \vdash_\Sigma e[z/x, z/y]{:}C} \qquad (\mathsf{LF}{:}\textsc{Share})$$

An LF program P consists of a signature Σ and a collection of terms e_f for each $f \in \mathrm{dom}(\Sigma)$ such that for all $f \in \mathrm{dom}(\Sigma)$ one has $y_1{:}A_1, \ldots, y_k{:}A_k \vdash_\Sigma e_f{:}C$ when $\Sigma(f) = (A_1, \ldots, A_k) \longrightarrow C$. In concrete examples we indicate the association of defining terms with function symbols by writing down equations of the form $f(y_1, \ldots, y_k) = e_f$.

We usually consider a fixed but arbitrary program P throughout the following.

We denote by $\mathsf{LF}^{\mathrm{lin}}$ the fragment of LF which neither contains the term constructor match' nor the typing rules LF:Share, LF:List-Elim'. Note that $\mathsf{LF}^{\mathrm{lin}}$ is an affine linear functional language.

2.1 Examples

Throughout the examples, the type A is any fixed (but arbitrary) LF-type. In an implemented version of LF one would presumably want to allow type variables and possibly even polymorphic quantification over these.

Example 1. The following example defines a function that reverses the order of the elements in a list of booleans.

$$\begin{aligned}
\mathtt{reverse} &: (\mathsf{L}(A)) \longrightarrow \mathsf{L}(A)\\
\mathtt{rev\_aux} &: (\mathsf{L}(A)\,, \mathsf{L}(A)) \longrightarrow \mathsf{L}(A)
\end{aligned}$$

$$\begin{aligned}
\mathtt{reverse}(l) &= \mathtt{rev\_aux}(l, \mathsf{nil})\\
\mathtt{rev\_aux}(l, acc) &= \mathsf{match}\ l\ \mathsf{with}\\
&\qquad \mid \mathsf{nil} \Rightarrow acc\\
&\qquad \mid \mathsf{cons}(h,t) \Rightarrow \mathtt{rev\_aux}(t, \mathsf{cons}(h, acc))
\end{aligned}$$

We furthermore define $\mathtt{reverse}'$ and $\mathtt{rev\_aux}'$ similarly, just replacing match by match'.

Example 2. The next example corresponds to the well-known insertion sort algorithm:

$$\begin{aligned}
\mathtt{sort} &: (\mathsf{L}(A)) \to \mathsf{L}(A)\\
\mathtt{ins} &: (A, \mathsf{L}(A)) \to \mathsf{L}(A)\\
\mathtt{leq} &: (A \otimes A) \to \mathsf{B} \otimes (A \otimes A)
\end{aligned}$$

$$\begin{aligned}
\mathtt{ins}(n,l) &= \mathsf{match}\ l\ \mathsf{with}\\
&\qquad \mid \mathsf{nil} \Rightarrow \mathsf{cons}(n, \mathsf{nil})\\
&\qquad \mid \mathsf{cons}(h,t) \Rightarrow\\
&\qquad\quad \mathsf{match}\ \mathtt{leq}(n,h)\ \mathsf{with}\ b \otimes \bigl(n' \otimes h'\bigr) \Rightarrow\\
&\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{cons}(n', \mathsf{cons}(h', t))\\
&\qquad\qquad\quad \mathsf{else}\ \ \mathsf{cons}(h', \mathtt{ins}(n', t))\\
\mathtt{sort}(l) &= \mathsf{match}\ l\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{nil}\\
&\qquad\qquad\qquad \mid \mathsf{cons}(h,t) \Rightarrow \mathtt{ins}(h, \mathtt{sort}(t))
\end{aligned}$$

To simplify notation we have used some syntactic sugar in these examples: notably we allow nesting of terms which expands into nested let-constructs and also allow nested patterns as in line 4 of $\mathtt{ins}$ which expand into a sequence of nested matches.

Here we assume the comparison function $\mathtt{leq}$ to return its arguments so that this example makes sense in the fragment $\mathsf{LF}^{\mathrm{lin}}$.

Example 3. The function $\mathtt{clone}$ doubles its input:

$$\begin{aligned}
\mathtt{clone} &: (\mathsf{L}(\mathsf{B})) \to \mathsf{L}(\mathsf{B}) \otimes \mathsf{L}(\mathsf{B})\\
\mathtt{clone}(l) &= \mathsf{match}\ l\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{nil} \otimes \mathsf{nil}\ \mid \mathsf{cons}(h,t) \Rightarrow\\
&\qquad \mathsf{match}\ \mathtt{clone}(t)\ \mathsf{with}\ t_1 \otimes t_2 \Rightarrow\\
&\qquad\quad \mathsf{if}\ h\ \mathsf{then}\ \mathsf{cons}(\mathsf{tt}, t_1) \otimes \mathsf{cons}(\mathsf{tt}, t_2)\\
&\qquad\qquad \mathsf{else}\ \ \mathsf{cons}(\mathsf{ff}, t_1) \otimes \mathsf{cons}(\mathsf{ff}, t_2)
\end{aligned}$$

Example 4. The function $\mathtt{tos}$ replaces each third element of a list by a value depending on its two predecessors, so it does not change the length of the list, but this implementation of $\mathtt{tos}$ is composed of two auxiliary functions, which *do* change the length of the list in between. Namely, $\mathtt{sec}$ deletes every third element whereas $\mathtt{tpo}$ inserts a new element in every third position.

The significance of the type $\mathsf{B} \otimes \mathsf{B}$ as opposed to B or an unspecified type will be explained in Section 7.

$$\begin{aligned}
\mathtt{tos} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B})) \longrightarrow \mathsf{L}(\mathsf{B} \otimes \mathsf{B})\\
\mathtt{sec} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B})) \longrightarrow \mathsf{L}(\mathsf{B} \otimes \mathsf{B})\\
\mathtt{tpo} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B})) \longrightarrow \mathsf{L}(\mathsf{B} \otimes \mathsf{B})
\end{aligned}$$

```
tos(l) = tpo(sec(l))
sec(l) = match l with
           | nil ⇒ nil
           | cons(h1, t1) ⇒ match t1 with
             | nil ⇒ cons(h1, nil)
             | cons(h2, t2) ⇒ match t2 with
               | nil ⇒ cons(h1, cons(h2, nil))
               | cons(h3, t3) ⇒ cons(h1, cons(h2, sec(t3)))
tpo(l) = match l with
           | nil ⇒ nil
           | cons(h1, t1) ⇒ match t1 with
             | nil ⇒ cons(h1, nil)
             | cons(h2, t2) ⇒
               cons(h1, cons(h2, cons(h1, tpo(t2))))
```

3. OPERATIONAL SEMANTICS

We use a freelist containing available heap cells. We treat this freelist simply as an integer value giving the number of free words.

Issues of alignment are assumed to be dealt with by an appropriate defragmentation routine to be launched whenever a request for t aligned words cannot be met although the freelist has size larger or equal than t. Admittedly, defragmentation is costly to implement. If desired, we can avoid fragmentation by assuming that all allocated blocks are of the same size. See also the remark on garbage collection at the end of this section.

Let Loc be a set of *locations* which model memory addresses on a heap abstracted over possible renaming that may become necessary upon defragmentation. We use ℓ to range over elements of Loc. Next we define a set of *values* Val, ranged over by v which occur as values of program variables, results, and values bound to locations in a heap.

$$v \;::=\; c \;\mid\; \ell \;\mid\; \mathsf{NULL} \;\mid\; (v,v) \;\mid\; \mathsf{inl}(v) \;\mid\; \mathsf{inr}(v)$$

A value is either a boolean constant c, a location ℓ, a null value NULL, a pair of values (v,v) or a value marked with either inl or inr. Occasionally we use a short hand notation for tuples, e.g. we write (v,v,v) instead of $(v,(v,v))$.

We assume that the LF type derivation is implicitly accessible (e.g. by adding a pointer to a type to each value as is done in Java), hence we allow ourselves to extend the size function to $\mathsf{SIZE} : \mathsf{Val} \to \mathbb{N}$. The idea is that value v occupies $\mathsf{SIZE}(v)$ words when stored in the heap. We are aware that this is not rigorous, however, the reduction on notational clutter outweighs the formal disadvantages by far.

A *stack* $S{:}\mathsf{Var} \rightharpoonup \mathsf{Val}$ is a finite partial mapping from variables to values, and a *heap* $\sigma{:}\mathsf{Loc} \rightharpoonup \mathsf{Val}$ is a finite partial mapping from locations to values. Evaluation of an expression e takes place with respect to a given stack and heap, and yields a value and a possibly updated heap. Moreover, the size of the freelist may shrink or grow upon evaluation. Thus we have a relation of the form $m, S, \sigma \vdash e \rightsquigarrow v, \sigma', m'$ expressing that the evaluation of e under stack S and heap σ succeeds in the presence of a freelist of size m and results in value v. As a side effect the heap is modified to σ' and the size of the freelist becomes m'. The values m and m' are arbitrary natural numbers.

The stack is extended with additional variable bindings whenever we enter a new scope, inside subterms in the premises of the evaluation rules. When we evaluate a function body we use a stack which only mentions the actual parameters, intuitively preventing access beyond the stack frame. Notice that the stack may contain pointers into the heap (i.e., locations), but there are no pointers going from the heap into the stack.

The operational semantics is given with respect to a fixed signature and program.

$$m, S, \sigma \vdash * \rightsquigarrow \mathsf{NULL}, \sigma, m \qquad (\rightsquigarrow_\Diamond{:}\textsc{Unit})$$

$$m, S, \sigma \vdash c \rightsquigarrow c, \sigma, m \qquad (\rightsquigarrow_\Diamond{:}\textsc{Const})$$

$$m, S, \sigma \vdash x \rightsquigarrow S(x), \sigma, m \qquad (\rightsquigarrow_\Diamond{:}\textsc{Var})$$

$$\frac{\begin{array}{c} S(x_1) = v_1 \ \cdots \ S(x_n) = v_n \\ m, [y_1{\mapsto}v_1, \ldots, y_n{\mapsto}v_n], \sigma \vdash e_f \rightsquigarrow v, \sigma', m' \\ \text{the } y_i \text{ are the symbolic arguments of } e_f \end{array}}{m, S, \sigma \vdash f(x_1, \ldots, x_n) \rightsquigarrow v, \sigma', m'} \ (\rightsquigarrow_\Diamond{:}\textsc{Fun})$$

$$\frac{\begin{array}{c} m, S, \sigma \vdash e_1 \rightsquigarrow v_1, \sigma_0, m_0 \\ m_0, S[x{\mapsto}v_1], \sigma_0 \vdash e_2 \rightsquigarrow v, \sigma', m' \end{array}}{m, S, \sigma \vdash \mathsf{let}\ \ x{=}e_1\ \ \mathsf{in}\ \ e_2 \rightsquigarrow v, \sigma', m'} \ (\rightsquigarrow_\Diamond{:}\textsc{Let})$$

$$\frac{S(x) \neq 0 \qquad m, S, \sigma \vdash e_t \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \rightsquigarrow v, \sigma', m'} \ (\rightsquigarrow_\Diamond{:}\textsc{If-T})$$

$$\frac{S(x) = 0 \qquad m, S, \sigma \vdash e_f \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \rightsquigarrow v, \sigma', m'} \ (\rightsquigarrow_\Diamond{:}\textsc{If-F})$$

$$m, S, \sigma \vdash x_1 \otimes x_2 \rightsquigarrow \big(S(x_1), S(x_2)\big), \sigma, m \ (\rightsquigarrow_\Diamond{:}\textsc{Pair})$$

$$\frac{S(x) = (v_1, v_2) \quad m, S[x_1{\mapsto}v_1][x_2{\mapsto}v_2], \sigma \vdash e \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ (x_1 \otimes x_2) \Rightarrow e \ \rightsquigarrow v, \sigma', m'}$$
$$(\rightsquigarrow_\Diamond{:}\textsc{Match-Pair})$$

$$m, S, \sigma \vdash \mathsf{inl}(x) \rightsquigarrow \mathsf{inl}\big(S(x)\big), \sigma, m \qquad (\rightsquigarrow_\Diamond{:}\textsc{Inl})$$

$$m, S, \sigma \vdash \mathsf{inr}(x) \rightsquigarrow \mathsf{inr}\big(S(x)\big), \sigma, m \qquad (\rightsquigarrow_\Diamond{:}\textsc{Inr})$$

$$\frac{S(x) = \mathsf{inl}(v') \qquad m, S[y{\mapsto}v'] \vdash e_1 \rightsquigarrow v, \sigma', m'}{\mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{inl}(y) \Rightarrow e_1 \mid \mathsf{inr}(y) \Rightarrow e_2 \rightsquigarrow v, \sigma', m'}$$
$$(\rightsquigarrow_\Diamond{:}\textsc{Match-Inl})$$

$$\frac{S(x) = \mathsf{inr}(v') \qquad m, S[y{\mapsto}v'] \vdash e_2 \rightsquigarrow v, \sigma', m'}{\mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{inl}(y) \Rightarrow e_1 \mid \mathsf{inr}(y) \Rightarrow e_2 \rightsquigarrow v, \sigma', m'}$$
$$(\rightsquigarrow_\Diamond{:}\textsc{Match-Inr})$$

$$m, S, \sigma \vdash \mathsf{nil} \rightsquigarrow \mathsf{NULL}, \sigma, m \qquad (\rightsquigarrow_\Diamond{:}\textsc{Nil})$$

$$\frac{v = \big(S(x_h), S(x_t)\big) \qquad \ell \notin \mathrm{dom}(\sigma)}{m + \mathsf{SIZE}(v), S, \sigma \vdash \mathsf{cons}(x_h, x_t) \rightsquigarrow \ell, \sigma[\ell{\mapsto}v], m}$$
$$(\rightsquigarrow_\Diamond{:}\textsc{Cons})$$

$$\frac{S(x) = \mathsf{NULL} \qquad m, S, \sigma \vdash e_1 \rightsquigarrow v, \sigma', m'}{m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with} \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(x_h, x_t) \Rightarrow e_2 \end{array} \rightsquigarrow v, \sigma', m'}$$
$$(\rightsquigarrow_\Diamond{:}\textsc{Match-Nil})$$

$$\frac{\begin{array}{c}S(x) = \ell \qquad \sigma(\ell) = (v_h, v_t) \qquad m_0 = m + \mathsf{SIZE}\,(\sigma(\ell)) \\ m_0, S[x_h{\mapsto}v_h][x_t{\mapsto}v_t], \sigma \setminus \ell \vdash e_2 \rightsquigarrow v, \sigma', m'\end{array}}{\begin{array}{l}m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \qquad\qquad \rightsquigarrow v, \sigma', m' \\ \qquad\qquad\qquad\qquad\qquad \mid \mathsf{cons}(x_h, x_t) \Rightarrow e_2\end{array}}$$
$$(\rightsquigarrow_{\Diamond}\text{:M}\textsc{atch}\text{-C}\textsc{ons})$$

$$\frac{S(x) = \mathsf{NULL} \qquad m, S, \sigma \vdash e_1 \rightsquigarrow v, \sigma', m'}{\begin{array}{l}m, S, \sigma \vdash \mathsf{match}'\ \ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \qquad\qquad \rightsquigarrow v, \sigma', m' \\ \qquad\qquad\qquad\qquad\qquad \mid \mathsf{cons}(x_h, x_t) \Rightarrow e_2\end{array}}$$
$$(\rightsquigarrow_{\Diamond}\text{:M}\textsc{atch}\text{'-N}\textsc{il})$$

$$\frac{\begin{array}{c}S(x) = \ell \qquad \sigma(\ell) = (v_h, v_t) \\ m, S[x_h{\mapsto}v_h][x_t{\mapsto}v_t], \sigma \vdash e_2 \rightsquigarrow v, \sigma', m'\end{array}}{\begin{array}{l}m, S, \sigma \vdash \mathsf{match}'\ \ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \qquad\qquad \rightsquigarrow v, \sigma', m' \\ \qquad\qquad\qquad\qquad\qquad \mid \mathsf{cons}(x_h, x_t) \Rightarrow e_2\end{array}}$$
$$(\rightsquigarrow_{\Diamond}\text{:M}\textsc{atch}\text{'-C}\textsc{ons})$$

The only rules that deserve an explanation are the ones pertaining to the match constructs for lists. It is assumed that the match construct immediately deallocates the node matched against, whereas it is preserved in a match′ construct. Accordingly the freelist grows in the branches of a match whereas it stays the same in a match′. At this point, the programmer decides which one to use. It is conceivable that this decision can be automated in such a way that the best possible resource behaviour is obtained. This is, however, left for future research.

Note that given m, S, σ, e it need not be the case that there exist v, σ', m' with $m, S, \sigma \vdash e \rightsquigarrow v, \sigma', m'$ for one of the following reasons:

- Non-termination (this manifests itself as an infinite backwards application of rule $\rightsquigarrow_{\Diamond}$:Fun)
- Wrong elements in stack or heap, e.g. a Boolean where either NULL or a pair is expected.
- Insufficiently large freelist, e.g. $m = 0, e = \mathsf{cons}(1, \mathsf{nil})$.

We choose to accept nontermination and rely on a standard typing discipline to deal with wrong elements. The main contribution here is to devise static methods that ensure absence of insufficiently large freelists.

We remark at this point that the judgement $m, S, \sigma \vdash e \rightsquigarrow v, \sigma', m'$ admits the following alternative interpretation. If we evaluate e using a garbage collector which collects after every pattern match then the heap size during the evaluation will not exceed the initial heap size by more than m.

3.1 Operational semantics without freelist

In order to be able to formally state correctness of the static analysis we are going to describe, it is convenient to introduce an auxiliary operational semantics which does not rely on freelists. To this end, we introduce a judgment $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ which intuitively reads as "in stack S and heap σ expression e evaluates to result v and leaves heap σ'". The rules defining this judgment are like the ones that define the instrumented judgment $m, S, \sigma \vdash e \rightsquigarrow v, \sigma', m'$ but without all reference to freelist sizes. For example, we have the rule

$$\frac{v = \big(S(x_h), S(x_t)\big) \qquad \ell \notin \mathrm{dom}(\sigma)}{S, \sigma \vdash \mathsf{cons}(x_h, x_t) \rightsquigarrow \ell, \sigma[\ell{\mapsto}v]} \ (\rightsquigarrow\text{:C}\textsc{ons})$$

We can understand this judgment as formalizing evaluation in a C-like environment where space is allocated whenever a cons-cell is formed and deallocated whenever we match against a cons-cell.

In earlier work [8, 2] it was shown that under a linear typing discipline, in particular in $\mathsf{LF}^{\mathrm{lin}}$, this judgment represents the intended functional semantics. In this paper, we will rely on the essence of these earlier results and do not speak about functional semantics at all. More precisely, we will establish a result of the following kind.

Correctness property. *If* $\Gamma \vdash_{\Sigma} e{:}A$ *in* LF *and our static analysis derives a minimum freelist size* n *then whenever* $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ *without malignant sharing then for all* $m \geq n$ *there exists* m' *such that* $m, S, \sigma \vdash e \rightsquigarrow v, \sigma', m'$.

3.2 Formalisation of benign sharing

We define a variant of the operational semantics: $S, \sigma \vdash e \rightsquigarrow^{\mathrm{bs}} v, \sigma'$ which differs from the original operational semantics in that it prohibits malignant sharing in the sense described in the Introduction.

The auxiliary function $\mathcal{R} : \text{heap} \times \mathsf{Val} \longrightarrow \mathcal{P}(\mathsf{Loc})$ is defined as follows:

$$\begin{array}{ll}\mathcal{R}(\sigma, c) = \emptyset & \mathcal{R}(\sigma, \mathsf{NULL}) = \emptyset \\ \mathcal{R}(\sigma, (v_1, v_2)) = \mathcal{R}(\sigma, v_1) \cup \mathcal{R}(\sigma, v_2) & \mathcal{R}(\sigma, \mathsf{inl}(v)) = \mathcal{R}(\sigma, v) \\ \mathcal{R}(\sigma, \ell) = \{\ell\} \cup \mathcal{R}(\sigma, \sigma(\ell)) & \mathcal{R}(\sigma, \mathsf{inr}(v)) = \mathcal{R}(\sigma, v)\end{array}$$

We set $\mathcal{R}(\sigma, \sigma(\ell)) := \emptyset$ when $\ell \notin \mathrm{dom}(\sigma)$. We extend $\mathcal{R}$ to stacks by: $\mathcal{R}(\sigma, S) := \bigcup_{x \in \mathrm{dom}\, S} \mathcal{R}\big(\sigma, S(x)\big)$. Intuitively, $\mathcal{R}(\sigma, S)$ is the set of locations accessible from S.

The judgment $S, \sigma \vdash e \rightsquigarrow^{\mathrm{bs}} v, \sigma'$ is now inductively defined by the rules for the ordinary (resource-free) operational semantics except for the rules $\rightsquigarrow$:Let and $\rightsquigarrow$:Match-Cons which are replaced by the following ones. The rules concerning match′ are not altered.

$$\frac{\begin{array}{c}S, \sigma \vdash e_1 \rightsquigarrow^{\mathrm{bs}} v_1, \sigma_0 \\ S[x{\mapsto}v_1], \sigma_0 \vdash e_2 \rightsquigarrow^{\mathrm{bs}} v, \sigma' \\ \sigma{\restriction}\mathcal{R}(\sigma, S') = \sigma_0{\restriction}\mathcal{R}(\sigma, S') \qquad S' = S{\restriction}\mathrm{FV}(e_2)\end{array}}{S, \sigma \vdash \mathsf{let}\ \ x{=}e_1\ \ \mathsf{in}\ \ e_2 \rightsquigarrow^{\mathrm{bs}} v, \sigma'} \ (\rightsquigarrow^{\mathrm{bs}}\text{:L}\textsc{et})$$

$$\frac{\begin{array}{c}S(x) = \ell \qquad \sigma(\ell) = (v_h, v_t) \\ \ell \notin \mathcal{R}\big(\sigma, S[x_h{\mapsto}v_h][x_t{\mapsto}v_t]{\restriction}\mathrm{FV}(e_2)\big) \\ S[x_h{\mapsto}v_h][x_t{\mapsto}v_t], \sigma \setminus \ell \vdash e_2 \rightsquigarrow^{\mathrm{bs}} v, \sigma'\end{array}}{S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \mid \mathsf{cons}(x_h, x_t) \Rightarrow e_2 \rightsquigarrow^{\mathrm{bs}} v, \sigma'}$$
$$(\rightsquigarrow^{\mathrm{bs}}\text{:M}\textsc{atch}\text{-C}\textsc{ons})$$

Since these rules have strengthened preconditions compared to their counterparts we clearly have

Lemma 1. $\sigma, S \vdash e \rightsquigarrow^{\mathrm{bs}} v, \sigma' \implies \sigma, S \vdash e \rightsquigarrow v, \sigma'$

Let us consider short program fragments illustrating malignant sharing: let $x{=}\mathtt{reverse}(y)$ in y, where `reverse` is defined as in Example 1. The function `reverse` reverses the list y destructively, hence the rule $\rightsquigarrow^{\mathrm{bs}}$:Let is not applicable, as y is contained in the reachable region and changes after evaluation of $\mathtt{reverse}(y)$. Note that the rule $\rightsquigarrow$:Let would go through. If the fragment would call `reverse′` instead, which produces a reversed copy via the use of match′ instead of match, the program fragment above would be acceptable. However, the difference would be revealed in the different resource consumption as will be shown in Section 4.1.

Now consider the fragment $\mathsf{let}\ x{=}y\ \mathsf{in}\ x +\!\!+\, y$, where the infix $+\!\!+$ denotes list concatenation (see definition in Example 7). Here the rule $\overset{\text{bs}}{\leadsto}$:LET would be applicable, but fails since $x +\!\!+\, y$ cannot be evaluated, unless $S(y) = \mathsf{NULL}$. The reason is that the evaluation of $x +\!\!+\, y$ deallocates x, but the locations reachable from x can also be reached via y, hence the precondition added to $\overset{\text{bs}}{\leadsto}$:MATCH-CONS is violated. Of course, we could define a copying version of "append" using match'. Note that our semantics does not cater for in place update. We can either create a new cell or deallocate a cell, but never change the contents of an existing cell. This precludes, in particular, the creation of circular data structures.

The annotated version $\overset{\text{bs}}{\leadsto}_{\Diamond}$ is formulated similarly, the resource related constraints do not change.

4. LF WITH RESOURCE ANNOTATIONS

In this section we introduce resource annotations for LF which will allow us to predict the amount of heap space needed to evaluate a program. This prediction will be a linear expression involving the sizes of the arguments.

We call this annotated version $\mathsf{LF}_\Diamond$. Accordingly, the linearly typed fragment not containing the rule $\mathsf{LF}_\Diamond$:SHARE and the match'-term constructors will be called $\mathsf{LF}_\Diamond^{\text{lin}}$.

The types of $\mathsf{LF}_\Diamond$ are given by the following grammar:

$$\begin{array}{rl}
\text{pure zero-order:} & P ::= \mathsf{1} \mid \mathsf{B} \mid P \otimes P \mid R + R \mid \mathsf{L}(R) \\
\text{rich zero-order:} & R ::= (P, k) \quad (\text{for } k \in \mathbb{Q}^+) \\
\text{first-order:} & F ::= (P, \ldots, P, k) \to R \quad (\text{for } k \in \mathbb{Q}^+)
\end{array}$$

The *underlying* LF-type $|A|$ of $\mathsf{LF}_\Diamond$-type A is defined by removing all resource annotations, for example $|\mathsf{L}(\mathsf{L}(\mathsf{B}, 5), 7)| = \mathsf{L}(\mathsf{L}(\mathsf{B}))$.

Furthermore we define $\mathsf{SIZE} : \mathsf{LF}_\Diamond\text{-type} \to \mathbb{N}$ by $\mathsf{SIZE}(A) := \mathsf{SIZE}(|A|)$, thus $\mathsf{SIZE}(A)$ does not depend on the annotations contained in A.

Let Σ be an $\mathsf{LF}_\Diamond$ signature mapping a finite set of function identifiers to $\mathsf{LF}_\Diamond$ first-order types, Γ be an $\mathsf{LF}_\Diamond$ typing context mapping a finite set of identifiers to $\mathsf{LF}_\Diamond$ pure zero-order types, and let n, n' be positive rationals. An $\mathsf{LF}_\Diamond$ typing judgment $\Gamma, n \vdash_\Sigma e{:}A, n'$ then reads "under signature Σ, in typing context Γ and with n memory resources available, the $\mathsf{LF}_\Diamond$ term e has type A with n' unused resources left over". In each of the following typing rules, let furthermore A, B, C denote arbitrary $\mathsf{LF}_\Diamond$ zero-order types and n, k, p, possibly decorated, denote arbitrary values in $\mathbb{Q}^+$.

$$\frac{n \geq n'}{\Gamma, n \vdash_\Sigma *{:}\mathsf{1}, n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Const Unit})$$

$$\frac{c \text{ a boolean constant} \qquad n \geq n'}{\Gamma, n \vdash_\Sigma c{:}\mathsf{B}, n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Const Bool})$$

$$\frac{x \in \mathrm{dom}(\Gamma) \qquad n \geq n'}{\Gamma, n \vdash_\Sigma x{:}\Gamma(x), n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Var})$$

$$\frac{\begin{array}{c}\Sigma(f) = (A_1, \ldots, A_p, k) \longrightarrow (C, k') \\ n \geq k \qquad n - k + k' \geq n'\end{array}}{\Gamma, x_1{:}A_1, \ldots, x_p{:}A_p, n \vdash_\Sigma f(x_1, \ldots, x_p){:}C, n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Fun})$$

$$\frac{\Gamma_1, n \vdash_\Sigma e_1{:}A, n_0 \qquad \Gamma_2, x{:}A, n_0 \vdash_\Sigma e_2{:}C, n'}{\Gamma_1, \Gamma_2, n \vdash_\Sigma \mathsf{let}\ \ x{=}e_1\ \mathsf{in}\ \ e_2{:}C, n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Let})$$

$$\frac{\Gamma, n \vdash_\Sigma e_t{:}A, n' \qquad \Gamma, n \vdash_\Sigma e_f{:}A, n'}{\Gamma, x{:}\mathsf{B}, n \vdash_\Sigma \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f{:}A, n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{If})$$

$$\frac{n \geq n'}{\Gamma, x_1{:}A_1, x_2{:}A_2, n \vdash_\Sigma x_1 \otimes x_2{:}A_1 \otimes A_2, n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Pair})$$

$$\frac{\Gamma, x_1{:}A_1, x_2{:}A_2, n \vdash_\Sigma e{:}C, n'}{\Gamma, x{:}A_1 \otimes A_2, n \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ x_1 \otimes x_2 \Rightarrow e{:}C, n'}$$
$$(\mathsf{LF}_\Diamond{:}\textsc{Pair-Elim})$$

$$\frac{n \geq k_l + n'}{\Gamma, x{:}A, n \vdash_\Sigma \mathsf{inl}(x){:}(A, k_l) + (B, k_r), n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Inl})$$

$$\frac{n \geq k_r + n'}{\Gamma, x{:}B, n \vdash_\Sigma \mathsf{inr}(x){:}(A, k_l) + (B, k_r), n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Inr})$$

$$\frac{\Gamma, y{:}A, n + k_l \vdash_\Sigma e_1{:}C, n' \qquad \Gamma, y{:}B, n + k_r \vdash_\Sigma e_2{:}C, n'}{\Gamma, x{:}(A, k_l) + (B, k_r), n \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{inl}(y) \Rightarrow e_1{:}C, n' \\ \mid \mathsf{inr}(y) \Rightarrow e_2 \end{array}}$$
$$(\mathsf{LF}_\Diamond{:}\textsc{Sum-Elim})$$

$$\frac{n \geq n'}{\Gamma, n \vdash_\Sigma \mathsf{nil}{:}\mathsf{L}(A, k), n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Nil})$$

$$\frac{n \geq \mathsf{SIZE}(A \otimes \mathsf{L}(A, k)) + k + n'}{\Gamma, x_h{:}A, x_t{:}\mathsf{L}(A, k), n \vdash_\Sigma \mathsf{cons}(x_h, x_t){:}\mathsf{L}(A, k), n'}$$
$$(\mathsf{LF}_\Diamond{:}\textsc{Cons})$$

$$\frac{\begin{array}{c}\Gamma, n \vdash_\Sigma e_1{:}C, n' \\ \Gamma, x_h{:}A, x_t{:}\mathsf{L}(A, k), n + \mathsf{SIZE}(A \otimes \mathsf{L}(A, k)) + k \vdash_\Sigma e_2{:}C, n'\end{array}}{\Gamma, x{:}\mathsf{L}(A, k), n \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \qquad\qquad :C, n' \\ \mid \mathsf{cons}(x_h, x_t) \Rightarrow e_2 \end{array}}$$
$$(\mathsf{LF}_\Diamond{:}\textsc{List-Elim})$$

$$\frac{\begin{array}{c}\Gamma, n \vdash_\Sigma e_1{:}C, n' \\ \Gamma, x_h{:}A, x_t{:}\mathsf{L}(A, k), n + k \vdash_\Sigma e_2{:}C, n'\end{array}}{\Gamma, x{:}\mathsf{L}(A, k), n \vdash_\Sigma \mathsf{match}'\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \qquad\qquad :C, n' \\ \mid \mathsf{cons}(x_h, x_t) \Rightarrow e_2 \end{array}}$$
$$(\mathsf{LF}_\Diamond{:}\textsc{List-Elim'})$$

$$\frac{\Gamma, x{:}A_1, y{:}A_2, n \vdash_\Sigma e{:}C, n'}{\Gamma, z{:}A_1 \oplus A_2, n \vdash_\Sigma e[z/x, z/y]{:}C, n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Share})$$

where $A_1 \oplus A_2$ is defined as follows when $|A_1| = |A_2|$:

$$\begin{array}{c}
\mathsf{1} \oplus \mathsf{1} = \mathsf{1} \qquad\qquad \mathsf{B} \oplus \mathsf{B} = \mathsf{B} \\
(A, k_1) \oplus (C, k_2) = \big(A \oplus C, k_1 + k_2\big) \\
(A_1 \otimes C_1) \oplus (A_2 \otimes C_2) = (A_1 \oplus A_2) \otimes (C_1 \oplus C_2) \\
(A_1 + C_1) \oplus (A_2 + C_2) = A_1 \oplus A_2 + C_1 \oplus C_2 \\
\mathsf{L}(A) \oplus \mathsf{L}(C) = \mathsf{L}(A \oplus C)
\end{array}$$

We observe that the following type rule is admissible:

$$\frac{\Gamma, n \vdash_\Sigma e{:}A, n_0 \qquad n' \leq n_0 + k}{\Gamma, n + k \vdash_\Sigma e{:}A, n'} \quad (\mathsf{LF}_\Diamond{:}\textsc{Waste})$$

If P is an $\mathsf{LF}_\Diamond$ program, then $|P|$ denotes the underlying LF program:

LEMMA 2. $\Gamma, n \overset{\mathsf{LF}_\Diamond}{\vdash_\Sigma} e{:}C, n' \implies |\Gamma| \overset{\mathsf{LF}}{\vdash_{|\Sigma|}} e{:}|C|$

4.1 Examples

We revisit the Examples presented in 2.1. Since the term languages of LF and $\mathsf{LF}_\Diamond$ are identical, we just give the proper $\mathsf{LF}_\Diamond$ signatures here. Again, A denotes a fixed pure $\mathsf{LF}_\Diamond$-type; let $a \in \mathbb{Q}^+$ be fixed (but arbitrary) as well.

Example 1.

$$\begin{aligned} \texttt{reverse} &: (\mathsf{L}(A,a)\,,0) \longrightarrow (\mathsf{L}(A,a)\,,0) \\ \texttt{rev\_aux} &: (\mathsf{L}(A,a)\,,\mathsf{L}(A,a)\,,0) \longrightarrow (\mathsf{L}(A,a)\,,0) \end{aligned}$$

While `reverse` reverses its input in place at no additional resource costs, `reverse`$'$ copies its argument so that it can be reused. For $a_0 = a + \mathsf{SIZE}\,(A \otimes \mathsf{L}(A)) = a + \mathsf{SIZE}\,(A) + 1$ we obtain the typing

$$\begin{aligned} \texttt{reverse}' &: (\mathsf{L}(A,a_0)\,,0) \longrightarrow (\mathsf{L}(A,a)\,,0) \\ \texttt{rev\_aux}' &: (\mathsf{L}(A,a_0)\,,\mathsf{L}(A,a)\,,0) \longrightarrow (\mathsf{L}(A,a)\,,0) \end{aligned}$$

In the explicit case $A = \mathsf{B}$ and $a = 0$ (hence $a_0 = 2$), we see that `reverse` can be computed without any additional resources, while `reverse`$'$ consumes $2n$ previously unused cells if run on an input list of length n (which itself already occupies $2n$ cells).

Example 2. Let again $a_0 = a + \mathsf{SIZE}\,(A) + 1$.

$$\begin{aligned} \texttt{sort} &: (\mathsf{L}(A,a)\,,0) \longrightarrow (\mathsf{L}(A,a)\,,0) \\ \texttt{ins} &: (A, \mathsf{L}(A,a)\,,a_0) \longrightarrow (\mathsf{L}(A,a)\,,0) \\ \texttt{leq} &: (A \otimes A, 0) \longrightarrow (\mathsf{B} \otimes (A \otimes A)\,,0) \end{aligned}$$

Example 3.

$$\texttt{clone} : (\mathsf{L}(\mathsf{B},2)\,,0) \longrightarrow (\mathsf{L}(\mathsf{B},0) \otimes \mathsf{L}(\mathsf{B},0)\,,0)$$

Example 4.

$$\begin{aligned} \texttt{tos} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B},0)\,,3) \longrightarrow (\mathsf{L}(\mathsf{B} \otimes \mathsf{B},0)\,,0) \\ \texttt{sec} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B},0)\,,3) \longrightarrow \left(\mathsf{L}\left(\mathsf{B} \otimes \mathsf{B}, \tfrac{3}{2}\right),0\right) \\ \texttt{tpo} &: \left(\mathsf{L}\left(\mathsf{B} \otimes \mathsf{B}, \tfrac{3}{2}\right),0\right) \longrightarrow (\mathsf{L}(\mathsf{B} \otimes \mathsf{B},0)\,,0) \end{aligned}$$

The intuition behind the fractional annotations will be explained in Section 7.

5. TRANSLATION TO LFPL

In [8] we have introduced a linear functional language that can be translated into C without dynamic memory allocation, i.e., without using the system calls malloc() and free().

This was achieved by introducing an abstract type $\Diamond$ standing for memory locations big enough to hold any structure node occurring in a particular program. Elements of this abstract type may be passed around as data, in particular they can arise as input, output, and components of structures. Constructors of recursive types take an extra argument of type $\Diamond$, e.g., $\mathsf{cons} : (\Diamond, A, \mathsf{L}(A)) \to \mathsf{L}(A)$. In the translation to C the space pointed to by this extra argument is used to store the newly create structure node. Conversely, in a pattern match we gain access to an element of type $\Diamond$ when matching against a recursive constructor such as cons. We will explain how $\mathsf{LF}_\Diamond^{\mathrm{lin}}$ can be used to infer LFPL-typings for $\mathsf{LF}^{\mathrm{lin}}$-programs.

Since LFPL handles resources as elements of type $\Diamond$ we restrict to integral annotations. For this purpose let $\mathsf{LF}_\Diamond^{\mathbb{N},\mathrm{lin}}$ denote the fragment of $\mathsf{LF}_\Diamond^{\mathrm{lin}}$where all annotations are restricted to nonnegative integers.

Furthermore, we temporarily redefine $\mathsf{SIZE}\,(A)$ to be 1 for all types A. This corresponds to the assumption made in LFPL that all structure nodes are stored in heap portions of equal size.

Types in $\mathsf{LF}_\Diamond^{\mathbb{N},\mathrm{lin}}$ can then be translated to LFPL-types by mapping each annotation n to an n-fold product of type $\Diamond$, for instance, the type $(A, \mathsf{L}(A,1)\,,2) \to (\mathsf{L}(A,1)\,,0)$ is mapped to $(A, \mathsf{L}(A \otimes \Diamond)\,, \Diamond \otimes \Diamond) \to (\mathsf{L}(A \otimes \Diamond))$.

The translation of terms follows the structure of a derivation in $\mathsf{LF}_\Diamond^{\mathbb{N},\mathrm{lin}}$; we omit the (essentially obvious) details.

This is useful since the resulting C-programs can be executed without overhead such as freelists, defragmentation, or garbage collection which makes them suitable in resource-restricted environments.

6. $\mathsf{LF}_\diamond$ AND SPACE-AWARE SEMANTICS

In this section we will prove a correspondence between full $\mathsf{LF}_\Diamond$ and the space-aware operational semantics from Section 3.

We must formalize that a given stack and heap fit a certain typing context:

$$\sigma \vdash \mathsf{NULL}{:}1 \qquad \text{(UNIT)}$$

$$\sigma \vdash c{:}\mathsf{B} \qquad \text{(BOOL)}$$

$$\frac{\sigma \vdash v{:}A_1 \qquad \sigma \vdash w{:}A_2}{\sigma \vdash (v,w){:}A_1 \otimes A_2} \qquad \text{(PAIR)}$$

$$\frac{\sigma \vdash v{:}A}{\sigma \vdash \mathsf{inl}(v){:}A + B} \qquad \text{(INL)}$$

$$\frac{\sigma \vdash v{:}B}{\sigma \vdash \mathsf{inr}(v){:}A + B} \qquad \text{(INR)}$$

$$\sigma \vdash \mathsf{NULL}{:}\mathsf{L}(A) \qquad \text{(LIST-NIL)}$$

$$\frac{\sigma \setminus \ell \vdash \sigma(\ell){:}A \otimes \mathsf{L}(A)}{\sigma \vdash \ell{:}\mathsf{L}(A)} \qquad \text{(LIST-NODE)}$$

$$\frac{\forall x_i \in \mathrm{dom}(\Gamma).\ \sigma \vdash S(x_i){:}\Gamma(x_i)}{\sigma \vdash S{:}\Gamma} \qquad \text{(CONTEXT)}$$

Furthermore we extend to $\mathsf{LF}_\Diamond$by

$$\sigma \vdash S{:}A_\Diamond \quad \Leftrightarrow \quad \sigma \vdash S{:}|A_\Diamond|$$

where $A_\Diamond$ is an $\mathsf{LF}_\Diamond$ type and similarly for contexts.

LEMMA 3. *Let σ, τ be heaps. If $\sigma \vdash v{:}A$ and $\forall \ell \in \mathcal{R}(\sigma, v).\sigma(\ell) = \tau(\ell)$ then $\tau \vdash v{:}A$*

Proofs of this and all subsequent propositions are contained in a technical report available on request and currently visible at our homepage.

LEMMA 4. *If $\Gamma \vdash_\Sigma e{:}A$ and $\sigma \vdash S{:}\Gamma\restriction FV(e)$ and $S,\sigma \vdash e \rightsquigarrow^{\mathrm{bs}} v,\sigma'$ then $\sigma' \vdash v{:}A$.*

We define $\Upsilon : \text{heap} \times \mathsf{Val} \times \mathsf{LF}\text{-type} \longrightarrow \mathbb{Q}^+$ by

$$\begin{aligned}
&\Upsilon(\sigma, v, \mathsf{1}) = \Upsilon(\sigma, c, \mathsf{B}) = 0\\
&\Upsilon(\sigma, (v_1, v_2), A \otimes B) = \Upsilon(\sigma, v_1, A) + \Upsilon(\sigma, v_2, B)\\
&\Upsilon(\sigma, \mathsf{inl}(v), (A,k) + (B,l)) = k + \Upsilon(\sigma, v, A)\\
&\Upsilon(\sigma, \mathsf{inr}(v), (A,k) + (B,l)) = l + \Upsilon(\sigma, v, B)\\
&\Upsilon(\sigma, \mathsf{NULL}, \mathsf{L}(A,k)) = 0\\
&\Upsilon(\sigma, \ell, \mathsf{L}(A,k)) = k + \Upsilon(\sigma, \sigma(\ell), A \otimes \mathsf{L}(A,k))
\end{aligned}$$

and furthermore $\Upsilon(\sigma, S, \Gamma) := \sum_{x \in \mathrm{dom}\,\Gamma} \Upsilon\big(\sigma, S(x), \Gamma(x)\big)$. The amount of additional heap space needed to evaluate a function $\mathtt{f} : (A_1, \dots, A_p, k) \to (B, k')$ depends on the size of the input to $\mathtt{f}$. If $\sigma \vdash S{:}\{x_1{:}A_1, \dots, x_p{:}A_p\}$, the amount of additional heap space required to compute $\mathtt{f}$ is $k + \Upsilon(\sigma, S, \{x_1{:}A_1, \dots, x_k{:}A_k\})$.

In particular, if $\mathtt{f} : (\mathsf{L}(\mathsf{B}, a), b) \to (\mathsf{L}(\mathsf{B}, c), d)$ then evaluating $\mathtt{f}(w)$ takes at most $a|w| + b$ extra space to evaluate, where $|w|$ is the length of w. If we evaluate $\mathtt{f}(w)$ given a freelist of size $a|w| + b + k$ (where $k \geq 0$) then after the evaluation the freelist will have size at least $c|\mathtt{f}(w)| + d + k$.

LEMMA 5. *If $\sigma\restriction\mathcal{R}(\sigma, v) = \sigma'\restriction\mathcal{R}(\sigma, v)$ then $\Upsilon(\sigma, v, A) = \Upsilon(\sigma', v, A)$.*

LEMMA 6. *For all σ, S, A_1, A_2, it holds that $\Upsilon(\sigma, v, A_1 \oplus A_2) = \Upsilon(\sigma, v, A_1) + \Upsilon(\sigma, v, A_2)$ provided that $\Upsilon(\sigma, v, A_1 \oplus A_2)$ is defined.*

THEOREM 1. *Let P be a valid $\mathsf{LF}_\Diamond$ program with signature Σ. For all $\mathsf{LF}_\Diamond$ terms e such that $\Gamma, n \vdash_\Sigma e{:}A, n'$ and whenever $S, \sigma \vdash e \rightsquigarrow^{\mathrm{bs}} v, \sigma'$ and $\sigma \vdash S : (\Gamma\restriction FV(e))$ then for all $q \in \mathbb{Q}^+$ and for all $m \in \mathbb{N}$ such that $m \geq n + \Upsilon(\sigma, S, \Gamma) + q$ there exists $m' \in \mathbb{N}$ satisfying $m' \geq n' + \Upsilon(\sigma', v, A) + q$ such that $m, S, \sigma \vdash e \rightsquigarrow^{\mathrm{bs}}_\Diamond v, \sigma', m'$.*

COROLLARY 1. *If P is a valid $\mathsf{LF}_\Diamond$ program containing a function symbol*

$$f : \big(\mathsf{L}(\mathsf{B}, n_1), \dots, \mathsf{L}(\mathsf{B}, n_k), m\big) \longrightarrow \big(\mathsf{L}(\mathsf{B}, n'), m'\big)$$

then the function call $f(l_1, \dots, l_k)$ evaluates properly to a list l', provided that there are at least $m + \sum_{i=1}^{k} n_i |l_i|$ free memory cells available, where $|l_i|$ denotes the number of nodes of list l_i. After the evaluation there are at least $m' + n'|l'|$ free cells available.

7. INFERENCE OF ANNOTATIONS

Recall that a *linear program* (LP) is a pair (V, C) where V is a set of variables and C is a set of inequalities of the form $a_1 x_1 + \dots a_n x_n \leq b$ where the x_i are variables from V and the a_i and b are rational numbers.

In addition, one may specify an *objective function* which is a term of the form $c_1 x_1 + \cdots + c_n x_n$ where the x_i are from V and the c_i are rational numbers. In this case, one defines an *optimal* solution to be a solution that minimizes the value of the objective function.

Our aim in this section is the following. Given an LF program P we want to discover whether there exists an $\mathsf{LF}_\Diamond$ program P' such that $|P'| = P$. To this end, we notice that the structure of any $\mathsf{LF}_\Diamond$-derivation is determined by its underlying LF-derivation.

This means that if we are given an LF-derivation of some program P all that needs to be done in order to obtain a corresponding $\mathsf{LF}_\Diamond$-derivation is to find the numerical values arising in type annotations in such a way that all the numerical side conditions are satisfied.

To discover these annotations, we assign to a given LF-program P (assumed to be equipped with a typing derivation) an LP $\mathcal{L}(P)$ with the property that solutions to $\mathcal{L}(P)$ are in 1-1 correspondence with $\mathsf{LF}_\Diamond$ programs P' such that $|P'| = P$. The LP $\mathcal{L}(P)$ is the pair (V, C) where V contains one specific variable for every occurrence of a numerical value in a possible $\mathsf{LF}_\Diamond$typing derivation.

The set C collects all the inequalities arising as side conditions in such a derivation. This includes in particular equality constraints that are implicit in that types are sometimes required to be equal, e.g. in rule $\mathsf{LF}_\Diamond$:VAR. Note that an equality constraint may be encoded as a pair of inequality constraints. Furthermore we add the constraints that all occurring variables are nonnegative, as all $\mathsf{LF}_\Diamond$-type annotations are nonnegative.

As an illustrative example, we consider a program P that contains a single function symbol $\mathtt{rev\_aux} : (\mathsf{L}(A), \mathsf{L}(A)) \to \mathsf{L}(A)$ with the defining expression as given in Example 1. We have the LF typing derivation shown in Figure 1.

In order to form $\mathcal{L}(P)$ we consider an "indeterminate" $\mathsf{LF}_\Diamond$-derivation as in Figure 2. It is clear that *any* $\mathsf{LF}_\Diamond$-derivation matching the LF-derivation of P arises as an instantiation of the derivation in Figure 2 satisfying the constraints given in Figure 3. Of course, we can readily eliminate all simple equality constraints given in Figure 3 leaving

$$\begin{aligned}
c &= n_2 - \mathsf{SIZE}(A) - 1 - b_1 & n_3 &\geq c\\
n_2 &\geq \mathsf{SIZE}(A) + 1 + b_2 + n_3 & n_3 - c + d &\geq d\\
c &\geq d
\end{aligned}$$

plus the nonnegativity constraints. Since we are only interested in the values of variables occurring within first-order types, we eliminate n_2, n_3 here in this example for a better understanding of the set of solutions and obtain:

$$c \geq d \geq 0 \qquad\qquad b_1 \geq b_2 = b_3 \geq 0$$

An optimal solution with respect to the sum of all variables is then given by $c = d = b_1 = b_2 = b_3 = 0$. Hence the typing $\mathtt{rev\_aux} : (\mathsf{L}(A, 0), \mathsf{L}(A, 0), 0) \to (\mathsf{L}(A, 0), 0)$ can be derived in $\mathsf{LF}_\Diamond$, which signifies that $\mathtt{rev\_aux}$ can be evaluated without any extra heap space.

These equations may also be regarded as the "most general $\mathsf{LF}_\Diamond$-type" of $\mathtt{rev\_aux}$, e.g. by $b_1 \geq b_2 = b_3$ we easily see that $\mathtt{rev\_aux}$ may also operate on lists containing an arbitrary amount of extra heap space, hence $\mathtt{rev\_aux} : (\mathsf{L}(A, 7), \mathsf{L}(A, 7), 0) \to (\mathsf{L}(A, 7), 0)$ could be derived if necessary by using $\mathtt{rev\_aux}$ in a more complicated program context.

The program from Example 4 portrays the usefulness of rational solutions. For the sake of simplicity we unify some variables which are obviously equated. We therefore assume the following enriched indeterminate signature:

$$\begin{aligned}
\mathtt{tos} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, l_1), x_1) \to (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, l_3), x_3)\\
\mathtt{sec} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, l_1), x_1) \to (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, l_2), x_2)\\
\mathtt{tpo} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, l_2), x_2) \to (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, l_3), x_3)
\end{aligned}$$

$$\dfrac{\dfrac{}{y{:}\mathsf{L}(A) \vdash y : \mathsf{L}(A)}\ \mathsf{LF}{:}\mathrm{VAR} \qquad \dfrac{\dfrac{}{y{:}\mathsf{L}(A)\,, h{:}A \vdash \mathsf{cons}(h,y) : \mathsf{L}(A)}\ \mathsf{LF}{:}\mathrm{CONS} \quad \dfrac{\Sigma(\texttt{rev\_aux}) = (\mathsf{L}(A)\,, \mathsf{L}(A)) \to \mathsf{L}(A)}{t{:}\mathsf{L}(A)\,, r{:}\mathsf{L}(A) \vdash \texttt{rev\_aux}(t,r) : \mathsf{L}(A)}\ \mathsf{LF}{:}\mathrm{FUN}}{y{:}\mathsf{L}(A)\,, h{:}A, t{:}\mathsf{L}(A) \vdash \mathsf{let}\ r{=}\mathsf{cons}(h,y)\ \mathsf{in}\ \texttt{rev\_aux}(t,r) : \mathsf{L}(A)}\ \mathsf{LF}{:}\mathrm{LET}}{x{:}\mathsf{L}(A)\,, y{:}\mathsf{L}(A) \vdash \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow y \mid \mathsf{cons}(h,t) \Rightarrow \mathsf{let}\ r{=}\mathsf{cons}(h,y)\ \mathsf{in}\ \texttt{rev\_aux}(t,r) : \mathsf{L}(A)}\ \mathsf{LF}{:}\mathrm{LIST\text{-}ELIM}$$

Figure 1: Derivation of P in LF

$$\dfrac{\dfrac{}{\begin{array}{l} y{:}\mathsf{L}(A,a_1)\,, n_1 \vdash \\ \quad y : \mathsf{L}(A,a_2)\,, m_1\end{array}}\ \mathsf{LF}_\Diamond{:}\mathrm{VAR} \qquad \dfrac{\dfrac{}{\begin{array}{l} y{:}\mathsf{L}(A,a_3)\,, h{:}A, n_2 \vdash \\ \quad \mathsf{cons}(h,y) : \mathsf{L}(A,a_4)\,, m_2\end{array}}\ \mathsf{LF}_\Diamond{:}\mathrm{CONS} \quad \dfrac{}{\begin{array}{l} t{:}\mathsf{L}(A,a_5)\,, r{:}\mathsf{L}(A,a_6)\,, n_3 \vdash \\ \quad \texttt{rev\_aux}(t,r) : \mathsf{L}(A,a_7)\,, m_3\end{array}}\ \mathsf{LF}_\Diamond{:}\mathrm{FUN}}{\begin{array}{l} y{:}\mathsf{L}(A,a_8)\,, h{:}A, t{:}\mathsf{L}(A,a_9)\,, n_4 \vdash \\ \quad \mathsf{let}\ r{=}\mathsf{cons}(h,y)\ \mathsf{in}\ \texttt{rev\_aux}(t,r) : \mathsf{L}(A,a_{10})\,, m_4\end{array}}\ \mathsf{LF}_\Diamond{:}\mathrm{LET}}{\begin{array}{l} x{:}\mathsf{L}(A,a_{11})\,, y{:}\mathsf{L}(A,a_{12})\,, n_5 \vdash \\ \quad \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow y \mid \mathsf{cons}(h,t) \Rightarrow \mathsf{let}\ r{=}\mathsf{cons}(h,y)\ \mathsf{in}\ \texttt{rev\_aux}(t,r) : \mathsf{L}(A,a_{13})\,, m_5\end{array}}\ \mathsf{LF}_\Diamond{:}\mathrm{LIST\text{-}ELIM}$$

where $\texttt{rev\_aux} : (\mathsf{L}(A,b_1)\,, \mathsf{L}(A,b_2)\,, c) \to (\mathsf{L}(A,b_3)\,, d)$. As an indeterminated $\mathsf{LF}_\Diamond$-type, A may contain further parameters.

Figure 2: Indeterminate derivation of P in $\mathsf{LF}_\Diamond$.

$$\begin{array}{ll}
a_1 = a_2, n_1 \geq m_1 & \mathsf{LF}_\Diamond{:}\mathrm{VAR} \\
a_3 = a_4, n_2 \geq \mathsf{SIZE}(A) + 1 + a_3 + m_2 & \mathsf{LF}_\Diamond{:}\mathrm{CONS} \\
a_5 = b_1, a_6 = b_2, a_7 = b_3, n_3 \geq c, n_3 - c + d \geq m_3 & \mathsf{LF}_\Diamond{:}\mathrm{FUN} \\
a_8 = a_3, a_9 = a_5, a_4 = a_6, a_{10} = a_7, & \\
n_4 = n_2, m_2 = n_3, m_3 = m_4 & \mathsf{LF}_\Diamond{:}\mathrm{LET} \\
a_{12} = a_1, a_{12} = a_8, a_{11} = a_9, a_{13} = a_2, a_{13} = a_{10}, & \\
n_5 = n_1, m_5 = m_1, n_5 = n_4 - \mathsf{SIZE}(A) - 1 - a_{11}, m_5 = m_4 & \mathsf{LF}_\Diamond{:}\mathrm{LIST\text{-}ELIM} \\
c = n_5, d = m_5, b_1 = a_{11}, b_2 = a_{12}, b_3 = a_{13} & \text{Valid program} \\
a_1, \ldots, a_{13}, b_1, \ldots, b_3, c, d, n_1, \ldots, n_5, m_1, \ldots m_5 \geq 0 & \text{Nonnegativity}
\end{array}$$

There may be further trivial constraints arising from the indeterminates in A.

Figure 3: Constraints of $\mathsf{LF}_\Diamond$-derivation in Figure 2

After simplification and elimination of all variables not occurring within the signature we are left with the following inequalities:

$$\begin{aligned}
&x_1 \geq x_2 \\
&x_1 \geq -(3 + l_1) + (3 + l_2) + x_2 \\
&x_1 \geq -2(3 + l_1) + 2(3 + l_2) + x_2 \\
&x_1 \geq -3(3 + l_1) + 2(3 + l_2) + x_1 - x_2 + x_2 \\
&x_2 \geq x_3 \\
&x_2 \geq -(3 + l_2) + (3 + l_3) + x_3 \\
&x_2 \geq -2(3 + l_2) + 3(3 + l_3) + x_2 - x_3 + x_3
\end{aligned}$$

plus nonnegativity constraints. A sensible solution to these inequalities is

$$\begin{aligned}
\texttt{tos} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, 0)\,, 3) \to (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, 0)\,, 0) \\
\texttt{sec} &: (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, 0)\,, 3) \to (\mathsf{L}\big(\mathsf{B} \otimes \mathsf{B}, \tfrac{3}{2}\big)\,, 0) \\
\texttt{tpo} &: (\mathsf{L}\big(\mathsf{B} \otimes \mathsf{B}, \tfrac{3}{2}\big)\,, 0) \to (\mathsf{L}(\mathsf{B} \otimes \mathsf{B}, 0)\,, 0)
\end{aligned}$$

Suppose we want to apply `tos` to the list l stored at ℓ in the heap σ having length $|l| = n$. This list occupies $3n$ heap cells (3 cells per node: a pair of booleans and one pointer, also see rule $\rightsquigarrow_\Diamond$:CONS). According to the type of `tos`, $0n + 3$ extra heap cells are required for evaluation (the additionally reserved heap space for l, which is $\Upsilon(\sigma, \ell, \mathsf{L}(\mathsf{B} \otimes \mathsf{B}, 0)) = 0$ plus 3 explicitly reserved cells). This amounts to $3n{+}3$ heap cells in total.

Now we first apply `sec` to l and call the resulting heap σ' Since `sec` destroys every third element of the list, $|\mathtt{sec}(l)| = \lceil \frac{2}{3} n \rceil$. Calculating the memory resources again, now according to the result type of `sec` yields: $3(\lceil \frac{2}{3} n \rceil) + \Upsilon(\sigma', \ell, \mathsf{L}\big(\mathsf{B} \otimes \mathsf{B}, \frac{2}{3}\big)) = 3(\lceil \frac{2}{3} n \rceil) + \frac{3}{2} \lceil \frac{2}{3} n \rceil \leq 3n + 3$. The memory cells freed by deleting list nodes of the input list allow an increase of additionally reserved heap space for the output list: Each deleted node frees three cells; as there are at least 2 remaining nodes per deleted node, the additional reserved heap space per node is $\frac{3}{2}$.

The inequality shows a possible memory leak of at most three cells in the case that l has length divisible by three. This is due to the fact that `sec` needs 3 additional cells to ensure the type $\mathsf{L}\big(\mathsf{B} \otimes \mathsf{B}, \frac{3}{2}\big)$ in the case that l has length $n = 3i + 2$ for some $i \in \mathbb{N}$. If the length is divisible by three, these extra resources are not needed, thus wasted.

We notice that the toplevel function `tos` also exhibits a "resource leak" since the three additional units required to call never show up in the result regardless of the length of the input. We remark that "deforestation", i.e., elimination of the intermediate result of the call to `sec` could overcome this. Whether this is an instance of a general pattern we cannot say at this point.

While it should be clear that fractional annotations describe the correct asymptotic behaviour one may wonder whether there might be problems with concrete inputs since, for example, allocating $\frac{3}{2}$ cells is not possible.

Consider a list l of length two, thus occupying 6 cells in view of $\mathsf{SIZE}\,(\mathsf{B} \otimes \mathsf{B} \otimes \mathsf{L}(\cdot)) = 3$. Applying `sec` to l returns an identical version of l and because of the annotation $\frac{3}{2}$ signals the availability of $3 = 2 \cdot \frac{3}{2}$ cells thus returning the three extra cells requested by `sec` in this case.

But now suppose that we match against this list; the rule $\mathsf{LF}_{\diamond}$:List-Elim then indicates the availability of $\frac{3}{2} + 3$ cells in the cons-branch. Of these, we can only use 4 immediately for storing operations on the heap. However, if we match again against the remaining part we gain access to the entire $9 = 6 + 3$ cells. Recall that $\mathsf{SIZE}\,(A) \in \mathbb{N}$.

8. INFERENCE FOR $\mathsf{LF}^{\mathbb{N},\mathrm{lin}}_{\diamond}$

In this section we consider the problem of inferring derivations in the fragment $\mathsf{LF}^{\mathbb{N},\mathrm{lin}}_{\diamond}$ from Section 5 which removes the sharing rule and restricts resource annotations to natural numbers. Clearly, such derivations for a given program P are in 1-1 correspondence to *integral* solutions of $\mathcal{L}(P)$.

As is well-known finding integral solutions of arbitrary LPs, let alone optimal ones, is an NP-hard problem.

However, we show that in a certain simplified subcase we can efficiently find integral solutions to $\mathcal{L}(P)$ that are optimal with respect to any objective function c whose coefficients are all nonnegative. As we want to minimize resource consumption, this is a sensible assumption on the objective function in the simplified subcase. Moreover, we show that in the general case finding integral solutions is again feasible whereas finding optimal solutions is NP-hard.

8.1 Inferring toplevel annotations

Suppose that we are only interested in solutions where all variables that occur within zero-order (sub-)types are zero as well as the variables occurring to the right hand side of first-order types.

In particular, we are looking at signatures of the form $(A_1, \ldots, A_\ell, n) \to (B, 0)$ where the A_i and B are $\mathsf{LF}_{\diamond}$-types with all annotations equal to zero.

Inspection of the typing rules then shows that after simplification of equality constraints the remaining system consists entirely of constraints of the form

$$x_0 \geq a_1 x_1 + a_2 x_2 + \cdots + a_\ell x_\ell + b$$

where the x_i are not necessarily distinct variables, the a_i are nonnegative integer coefficients, and b is an arbitrary integer constant. The only typing rules which might produce inequalities not of this form are $\mathsf{LF}_{\diamond}$:Fun, $\mathsf{LF}_{\diamond}$:Sum-Elim, $\mathsf{LF}_{\diamond}$:List-Elim, but we know that here the problematic negative variables (i.e. those occurring positively on the right hand side of the $\geq$ or negatively on the left hand side) are all zero by the assumption made in the simplified case. We call such a constraint *almost positive*.

Theorem 2. *Let $(\{x_1, \ldots, x_d\}, C)$ be an LP where C consists entirely of almost positive constraints. Let $c_1, \ldots, c_d \in \mathbb{N}$. The optimal integral solution of this LP with respect to the objective function $c_1 x_1 + \ldots c_d x_d$ can be found in polynomial time.*

To prove this one shows that the optimal rational solution is necessarily integral.

For an example we consider the LP arising from Example 2. In the enriched signature there are only three variables remaining in the simplified case:

$$\begin{aligned} \texttt{sort} &: (\mathsf{L}(A,0)\,, x_s) \to (\mathsf{L}(A,0)\,, 0) \\ \texttt{ins} &: (A, \mathsf{L}(A,0)\,, x_i) \to (\mathsf{L}(A,0)\,, 0) \\ \texttt{leq} &: (A \otimes A, x_l) \to (\mathsf{B} \otimes (A \otimes A)\,, 0) \end{aligned}$$

We do not give a concrete implementation of `leq` here and just assume that a call to `leq` does not require any resources. Therefore we immediately set $x_l := 0$ throughout this example. The actual value of $\mathsf{SIZE}\,(A)$ is unimportant.

Now we derive the LP as usual, inserting 0 whenever a new numerical value is needed within an $\mathsf{LF}_{\diamond}$ zero-order type or in the right-hand side of a first-order type.

After simplifying we are left with four almost positive constraints:

$$\begin{array}{ll} x_i \geq \mathsf{SIZE}\,(A) + 1 & \qquad x_s \geq 0 \\ x_i \geq 2x_i - (\mathsf{SIZE}\,(A) + 1) & \qquad x_i \geq 0 \end{array}$$

hence $x_s = 0$ and $x_i = \mathsf{SIZE}\,(A) + 1$ would be the optimal solution for any objective function $c_1 x_s + c_2 x_i$ with $c_1, c_2 \geq 0$.

Many more programs fall under the simplified subcase. This includes the quicksort example in Section 9 and all the LFPL-examples contained in [8].

We remark that setting the annotations contained in types and in result positions to *fixed* values other than zero also leads to almost positive LPs.

8.2 Efficient solutions for the general case

Let us call an LP *almost conical* if all inequalities are of one of the following two forms:

$$a_1 x_1 + \cdots + a_\ell x_\ell \leq 0 \qquad\qquad x \geq b$$

where $a_i \in \mathbb{Z}$ and $b \in \mathbb{N}$.

In this case, the set of rational solutions is closed under multiplication with scalars $\lambda \geq 1$. Therefore, we can obtain an integer solution from a rational solution by multiplying with the least common denominator.

We now show that for any $\mathsf{LF}^{\mathrm{lin}}$-program P the LP $\mathcal{L}(P)$ can be transformed into an almost conical one by performing a substitution of variables. Solving the resulting system and substituting back then yields a solution of $\mathcal{L}(P)$.

We observe that the only places where constants different from zero are introduced into constraints is via $\mathsf{SIZE}\,(\cdot)$ in the rules $\mathsf{LF}_{\diamond}$:Cons, $\mathsf{LF}_{\diamond}$:List-Elim.

The nonzero constants of the form $\mathsf{SIZE}\,(A)$ always occur together with the variable measuring the resource content of the corresponding list type. Hence, grouping these constants together with their associated variables by a substitution can be shown to yield an almost conical system.

Theorem 3. *Let P be a valid $\mathsf{LF}^{\mathrm{lin}}$-program then there exists an almost conical ILP (V, C) and a nonnegative integer vector c such that the solution set of $\mathcal{L}(P)$ is equal to $\{x - c \mid x \text{ solves } C\}$.*

We remark that this result does not hold in the presence of rules $\mathsf{LF}_{\diamond}$:Share and $\mathsf{LF}_{\diamond}$:List-Elim'.

Corollary 2. *There exists a polynomial time algorithm that given a valid* $\mathsf{LF}^{\mathrm{lin}}$*-program* P *determines a solution of* $\mathcal{L}(P)$ *if one exists and reports failure otherwise.*

Reconsidering Example 4 with this method yields:

$$\begin{aligned}
\texttt{tos} &: (\mathsf{L}(\mathsf{B}\otimes\mathsf{B},3)\,,6) \to (\mathsf{L}(\mathsf{B}\otimes\mathsf{B},3)\,,0)\\
\texttt{sec} &: (\mathsf{L}(\mathsf{B}\otimes\mathsf{B},3)\,,6) \to (\mathsf{L}(\mathsf{B}\otimes\mathsf{B},6)\,,0)\\
\texttt{tpo} &: (\mathsf{L}(\mathsf{B}\otimes\mathsf{B},6)\,,0) \to (\mathsf{L}(\mathsf{B}\otimes\mathsf{B},3)\,,0)
\end{aligned}$$

We note that this integral solution requires additional resources three times the length of the input list, which are finally left over after computation, whereas the fractional solution shows that these are unnecessary as can also be seen by merging the definitions of `tpo` and `sec` into specific optimized linear functional code for `tos`.

Although there are other integral solutions for this example, the presented solution is (under certain aspects) the best integral solution. However we cannot guarantee this. While finding a solution to an almost conical LP is feasible, finding an optimal solution is not:

Theorem 4. *For every instance* Φ *of 3SAT with* m *variables we can find an almost conical LP and an objective function so that a solution of objective value* $\leq n$ *exists iff* Φ *is satisfiable.*

Moreover, it was shown in [11] that such ILPs may indeed arise from inference problems. Hence we have:

Corollary 3. *Let* P *be a valid* LF *program. Finding an optimal solution of* $\mathcal{I}(P)$ *with respect to a given, arbitrary objective function is an NP-hard task.*

9. EXAMPLES

In this section we collect several illustrative examples.

Example 5. We demonstrate that the Quicksort algorithm falls within the simplified subcase presented in Section 8.1:

$$\begin{aligned}
\texttt{qsort} &: (\mathsf{L}(A,0)\,,0) \longrightarrow \mathsf{L}(A,0)\\
\texttt{split\_by} &: (A,\mathsf{L}(A,0)\,,0) \longrightarrow \mathsf{L}(A,0)\otimes\mathsf{L}(A,0)\\
\text{infix} \leq &: (A\otimes A,0) \to (\mathsf{B},0)
\end{aligned}$$

```
      qsort(l) = match l with
                  | nil ⇒ nil
                  | cons(h, t) ⇒
                      match split_by(h, t) with u ⊗ l ⇒
                       qsort(u) ++ cons(h, nil) ++ qsort(l)
split_by(p, l) = match l with
                  | nil ⇒ nil ⊗ nil
                  | cons(h, t) ⇒
                      match split_by(p, t) with u ⊗ l ⇒
                        if h ≤ p then cons(h, u) ⊗ l
                                 else  u ⊗ cons(h, l)
```

Please note that the standard functional implementation of quicksort, using a filtering function twice with mutually exclusive filter conditions instead of `split_by`, has no valid $\mathsf{LF}_\Diamond$-derivation. Calling the filter twice requires the duplication of the input list, while the type information is not enough to deduce that the filter cuts down each copy so that the sum of the lengths of each list is equal to the original list.

The sharing of heap-allocated data structures may simulate a duplication in some situations, but this of course restricts the use to read-only access (except for the last access) in order to prevent malignant sharing.

The following two examples show a sensible use of sharing and hence rely on rule $\mathsf{LF}_\Diamond$:Share; their evaluation exhibits no malignant sharing on all possible inputs so that Theorem 1 applies.

Example 6. For calculating the length of a list it is convenient to assume a type representing a finite part of the natural numbers and the presence of the usual arithmetic functions, e.g. $\mathsf{N} := \mathsf{B}^{\otimes 32}$.

$$\texttt{length} : (\mathsf{L}(A,0)\,,0) \to (\mathsf{N},0)$$

```
length(l) = match′ l with
               | nil ⇒ 0
               | cons(h, t) ⇒ 1 + length(t)
```

Example 7. While the length of a list could still be computed in $\mathsf{LF}^{\mathrm{lin}}_\Diamond$ without destroying the list (`length` might immediately rebuild the input list and return it together with the value for the length) at the cost of inconvenient programming, the following example exhibits proper sharing of heap-allocated data structures.

This example uses a type $\mathsf{T}(A)$ of binary trees whose internal nodes are labelled with A; leaves are unlabelled and represented by NULL. Its annotated version is $\mathsf{T}(A,k)$. We have $\Upsilon(\sigma,\mathsf{NULL},\mathsf{T}(A,k)) = 0$ and $\Upsilon(\sigma,\ell,\mathsf{T}(A,k)) = k + \Upsilon(\sigma,\sigma(\ell),A\otimes\mathsf{T}(A,k)\otimes\mathsf{T}(A,k))$. Thus, the amount of resource associated with such a tree is k times the number of its internal nodes.

$$\begin{aligned}
\texttt{pathlist} &: (\mathsf{T}(A,1)\,,2) \to (\mathsf{L}(\mathsf{L}(A,0)\,,0)\,,0)\\
\texttt{pathacc} &: (\mathsf{T}(A,1)\,,\mathsf{L}(A,0)\,,2) \to (\mathsf{L}(\mathsf{L}(A,0)\,,0)\,,0)\\
\text{infix} +\!\!+ &: (\mathsf{L}(C,a)\,,\mathsf{L}(C,a)\,,0) \to (\mathsf{L}(C,a)\,,0)
\end{aligned}$$

As we referred to ++ a few times, we present here a generic version. For this example it suffices to set $C = \mathsf{L}(A,0)$ and $a := 0$.

```
pathlist(t) = pathacc(t, nil)
pathacc(t, c) = match t with
                 | leaf ⇒ cons(c, nil)
                 | node(a, l, r) ⇒ let x=cons(a, c) in
                     pathacc(l, x) ++ pathacc(r, x)
     ++(l, r) = match l with
                 | nil ⇒ r
                 | cons(h, t) ⇒ cons(h, t ++ r)
```

The function `pathlist` turns a tree into a list of lists of booleans which contains the path from each leaf to the root. The nodes of the sublists (one for each leaf) are aliased among each other, thereby mimic the exact structure of the former tree within the heap, saving an exponential amount of space.

10. RELATED WORK

Approaches based on abstract interpretation and symbolic evaluation [7, 13, 4, 20, 5, 6] go in the direction of the naive approach mentioned in the Introduction. The structure of the inferred resource bound matches the structure of the program. Where the program contains a while loop or a recursion the bounding function will do so as well. This is not meant to diminish the value of those works: To begin the abstract interpretation removes useless computation so that computing the bound v will in general be easier than running f itself. This can greatly simplify profiling and testing. Furthermore, in many cases the recurrences reminiscent of iteration constructs in the original code can be solved using various methods from computer algebra.

What distinguishes our approach from these is that the resulting linear bounds once established are trivial to evaluate for concrete input lengths, that they are independently verifiable and that the algorithm for their intention is provably successful and efficient in a well-delineated subset of programs which comprises most textbook examples of functional programming such as reversal, quicksort, insertion sort, heap sort, Huffman codes, tree traversal, etc. Indeed, Unnikrishnan et al. [20] report performance problems with medium-sized inputs and recommend to fit an algebraic expression into a value table obtained from small inputs. This is acceptable for profiling purposes but certainly not for resource certification.

In other works like [3] the user must provide a conjectured resource bound. The formalism can be used to validate it but even for the validation user interaction is required. Moreover, this work only accounts for execution time not heap space.

Another piece of well-known related work are Hughes and Pareto's *sized types* [10]. This system allows one to certify upper bounds on the number of constructor symbols in inductive data types. For example List k A is the type of Lists of type A of length at most k, and accordingly "append" has the type List k_1 $A \to$ List $(k_2 + 1)$ $A \to$ List $(k_1 + k_2)$ A. A comparison to the type of the append function ++ from Example 7 reveals the different use of the annotations: While the annotation of sized lists yields upper bounds on the length, our annotation is a multiplicative constant which does not restrict the length of lists of this type. The approaches are thus quite different technically.

Nevertheless, sized types can also be used to infer space bounds. The transition from size to space is made via region-based memory management [19] which however, imposes unnatural restrictions due to the fact that a given data structure, e.g. a list, must reside entirely in one region. This prevents the analysis of computations in which lifetimes of data structures overlap, e.g. in the insertion sort algorithm according to §5.7 of [10]. The authors speculate on a possible solution based on *region resetting* and liveness inference, but this is not worked out in [10] nor in the later [16]. We emphasize that proper dynamic memory allocation is not modelled in [10]. This is acceptable in view of the intended application of sized types to embedded programming, but not—in our opinion—in a general functional programming context.

Another possible advantage of inferring space bounds directly, as we do, could lie in improved efficiency: Merely checking sized type requires Presburger Arithmetic (complete for doubly exponential time) compared to the polynomial time LP that we use. In this regard it would of course be interesting to know the exact complexity of sized type checking; more mundanely, whether the full strength of Presburger Arithmetic is really needed for this problem. The feasibility of *inference* as opposed to checking is left unanswered in [16, 10].

Unlike [10] and [5] we do not analyse stack size in this paper. We think that the linear bounds on stack size are often not adequate since typical algorithms can either be optimised using tail recursion to use constant stack or use a stack of logarithmic size, e.g. divide-and-conquer methods.

Furthermore, our system naturally encompasses trees, lists of trees, etc., whereas sized types seem to work primarily for linear data structures. While trees appear in the formal presentation in [16] none of the examples uses them; not even the type of the constructor for trees appears explicitly.

On the other hand, [16] contains a detailed and interesting account of infinite lists (streams). An exploration of streams in our framework must be left to further research.

11. CONCLUSIONS

We have presented an efficient and automatic analysis of heap usage of first-order functional programs. While we find that our analysis is surprisingly versatile and accurate there are a number of ways in which it can be improved.

Our analysis sometimes gives too modest assumptions about the memory available after execution of a function. A typical example is $\texttt{flatten} : \mathsf{L}(\mathsf{L}(A)) \to \mathsf{L}(A)$ assumed to be the natural implementation of flattening on lists of lists. Calling $\texttt{flatten}(w)$ returns $|w|$ heap space. However, our system assigns for example the type $\mathsf{L}(\mathsf{L}(A, 0), 0) \to \mathsf{L}(A, 0)$ hence not notifying the net resource-gain.

To fix this particular case it is tempting to introduce some kind of dependent typing allowing one to refer to the size or length of the input in the cost term of the result position. However, developing such a system whilst maintaining guarantees on efficient solvability is a delicate matter and must be left for future research.

As it stands, the system is sometimes insufficiently polymorphic. Namely, it can happen that two usages of an already defined function require two different annotations. Even if both these annotations are compatible with the definition of f only one of them can actually be assigned in $\mathsf{LF}_\diamond$. Consider, for instance, the identity function $\texttt{f} : \mathsf{L}(B) \to \mathsf{L}(B)$ defined by $\texttt{f}(x) = x$. In $\mathsf{LF}_\diamond$ we must assign a particular type, say $\mathsf{L}(B, 5), 3 \to \mathsf{L}(B, 5), 3$. In this case, we are not able to apply f to an argument of type $\mathsf{L}(B, 0)$.

To address this problem within the framework of the given system we can split a program into blocks of mutually dependent functions and perform the analysis separately for each of the blocks of definition. When using a function f outside its block of definition we can consider the entire LP of function f's definition rather than a particular solution. This approach can be seen as a definitional extension if we consider each occurrence of f outside its defining block as the usage of an identical copy of f.

If we also want to enable *polymorphic recursion*, i.e., a different instantiation of constraint variables in every recursive call, we must replace $\mathsf{LF}_\diamond$ with a constrained type system whose judgments are of the form $\mathcal{C}, \Gamma, n \vdash e{:}A, n$ where Γ, A, m, n may contain variables and $\mathcal{C}$ is a set of linear inequalities constraining these. The details are left for future work, but appear to be within the reach of the methods developed here.

A similar issue arises with higher-order functions. Simple use of higher-order functions merely as a means for modularization such as in combinators like `map`, `filter`, etc. can be accommodated by introducing several definitions, one for each usage, possibly hidden under some appropriate syntactic sugar. Formally, this kind of usage of higher-order functions is the one supported by the C language: the only expressions of function types are variables and constants.

If we aim for more general function expressions like partially-applied functions and lambda expressions as in functional programming languages the problem of heap space inference becomes much more complicated as we need to monitor the size of closures which are much more dependent on dynamic aspects. This is discussed in some detail in [9]. We do not see at this point how our work could be extended to cover general higher-order functions, not even linear ones. One referee suggested to investigate Reynolds' idea of *defunctionalisation* [17] which eliminates closures in favour of sum types. Again, we leave this to future work.

12. REFERENCES

[1] Mobile resource guarantees. EU Project No. IST-2001-33149, see `http://www.dcs.ed.ac.uk/home/mrg/`.

[2] David Aspinall and Martin Hofmann. Another Type System for In-Place Update. In D. Le Metayer, editor, *Programming Languages and Systems (Proc. ESOP'02)*, volume Springer LNCS 2305, 2002.

[3] K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th Symp. Principles of Prog. Lang. (POPL)*, pages 184–198. ACM, 2000.

[4] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-Upsilon-Omega: An assistant algorithms analyzer. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, pages 201–212, 1989. Proceedings AAECC'6, Rome, July 1988.

[5] Gustavo Gómez and Yanhong A. Liu. Automatic accurate cost-bound analysis for high-level languages. In Frank Mueller and Azer Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES'98, Montreal, Canada*. Springer, 1998. LNCS 1474.

[6] Gustavo Gómez and Yanhong A. Liu. Automatic time-bound analysis for a higher-order language. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 75–86. ACM Press, 2002.

[7] Bernd Grobauer. *Topics in Semantics-based Program Manipulation*. PhD thesis, BRICS Aarhus, 2001.

[8] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000. An extended abstract has appeared in *Programming Languages and Systems*, G. Smolka, ed., Springer LNCS, 2000.

[9] Martin Hofmann. The strength of non size-increasing computation. 2002. Proc. ACM Symp. on Principles of Programming Languages (POPL), Portland, Oregon.

[10] J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ML programming. In *Proc. International Conference on Functional Programming (ACM). Paris, September '99.*, pages 70–81, 1999.

[11] Steffen Jost. Static prediction of dynamic space usage of linear functional programs, 2002. Diploma thesis at Darmstadt University of Technology, Department of Mathematics. Available at `www.tcs.informatik.uni-muenchen.de/~jost/da_sj_28-02-2002.ps`.

[12] Naoki Kobayashi. Quasi-linear types. In *Proceedings ACM Principles of Programming Languages*, pages 29–42, 1999.

[13] H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, 1998.

[14] George Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Prog. Lang. (POPL)*. ACM, 1997.

[15] Martin Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 390–407. Springer-Verlag, February 1992. Lecture Notes in Computer Science 582.

[16] Lars Pareto. *Types for crash prevention*. PhD thesis, Chalmers University, Göteborg, Sweden, 2000.

[17] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, 1972.

[18] Natarajan Shankar. Efficiently executing PVS. Technical report, Computer Science Laboratory, SRI International, 1999.

[19] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[20] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Automatic accurate live memory analysis for garbage-collected languages. In *Proceedings of The Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001), June 22-23, 2001 / The Workshop on Optimization of Middleware and Distributed Systems (OM 2001), June 18, 2001, Snowbird, Utah, USA.*