Technische Universität Darmstadt
Fachbereich Mathematik

Static prediction of dynamic space usage of linear functional programs

Steffen Jost

supervised by

Prof. Dr. Martin Hofmann

Ludwig Maximilians Universität München

Lehrstuhl Theoretische Informatik

February 2002

Abstract

In [Hof00], Martin Hofmann introduced a first-order linear functional programming language called LFPL that allows one to read off a program's dynamic space usage from its signature. In this work we will settle the question *"To what extent can an ordinary linear functional program be translated automatically to* LFPL*, thus enabling us to infer the program's heap space usage?"*, which was already posed by Martin Hofmann in his work.

Our approach is as follows: By a static analysis of the given linear functional program's type derivation we construct an integer linear program (ILP) that is solvable if and only if a translation into LFPL exists. We describe the construction of the LFPL program once a solution to the assorted ILP is known. Furthermore we also show that the constructed ILPs are solvable in polynomial time. Finally we discuss an attempt to eliminate the restriction that the given program must be linearly typed.

Acknowledgments

I would like to express my gratitude to everybody who helped me to write the diploma thesis now in your hands:

First I would like to thank Prof. Dr. Martin Hofmann who guided me very well on my way to write this thesis and for pushing me ahead when my pessimistic estimates on the remaining working time slowed me down at last.

I would also like to thank Prof. Dr. Thomas Streicher for pointing out to me that Martin had a task that would neatly fit with the mixture of topics I had messed around with in my studies so far and for sharing his office for phone calls to Martin.

Thanks to the friendly family Kink who provided shelter for me during the two month of my first stay in the overcrowded city of Munich.

For the nice reception I received at the LMU and for helping me to find my way around (and through many locked doors), I would like to thank all the friendly people of TCS and PMS at the LMU in Munich.

Thanks to Klaus Aehlig for proofreading within very limited time and giving me very valuable remarks; I hope that this final version now pleases you well.

Thanks for proofreading and for keeping me awake in the night before my final examination at Darmstadt in order to discuss the principles of functional programming to Steffen Wendeberg.

Thanks to my girlfriend Andrea for proofreading through a work concerned with an completely unfamiliar field of mathematics and for providing sunshine when clouds grew too dark.

And finally a lot of thanks to my family and all my old and new friends both in Darmstadt and Munich who supported me in many quite different (and sometimes subtle) ways...

Contents

Introduction

Functional programming languages allow swift mathematically oriented implementations and generate programs that are more easily verified than imperative code. While a program of a classical imperative language like C must be viewed as an ordered set of instructions (e.g. store the value 5 in variable x; increment x by 1; goto step 17; ...), functional programs can be considered as nothing more than a (recursive) definition of a function from some input to some output. Hence one of their advantages is that the programmer is relieved from any issues concerning the storage of run-time values. One of their disadvantages is therefore that any implementation usually excessively wastes memory resources and depends upon an automatic Garbage Collection[1] for salvaging. Since even provably correct code may fail due to physical memory restrictions, it is therefore desirable to estimate bounds on the space consumption prior to program execution. Many work has been done so far to address this problem:

- Atsushi Igarashi and Naoki Kobayashi [I&K00] showed a way to an improved Garbage Collection via the use of a linear type system,[2] but still lack concrete bounds on the memory consumption. Linearity of types is not strictly enforced, the system is able to distinguish between linear types that are used at most once and non-linear ones which may be accessed several times.

- Karl Crary and Stephanie Weirich [C&W00] implemented a type system producing executables with certifiable bounds on running time depending on the input. The system requires the programmer to specify input-depending arithmetic expressions for function types and verifies whether or not the program will terminate before a virtual counter set to these depending values reaches zero. Once the specification is certified it is not needed at run time anymore. They conjecture that the technique can be generalised to certify bounds on space consumption (both on stack- and heap-space[1]).

- Mads Tofte and Jean-Pierre Talpin [T&T97] aim at eliminating the need for a Garbage Collection; they specify a Region Inference System, which infers the lifetimes of data structures. Memory is allocated and deallocated in parts called regions, which can be independently used for storing a multitude of values. The lifetime of a value then depends on the lifetime of the region it belongs to; while the lifetime of a region is determined by the program block it spans.

[1] see A.1 for a reference on commonly used notions throughout this work

[2] Variables may be used at most once in a linearly typed functional language. Data-objects are considered as destroyed after one-time use; also see A.1.

- John Hughes and Lars Pareto [H&P99] give a type system for programs running in constant space (both on stack- and heap-space). Their approach, a variant of the aforementioned Region Inference System of Tofte and Talpin, also guarantees termination, which is desirable for highly sensitive programs, but must also restrict the expressive power.

- Alan Mycroft and Richard Sharp [M&S00] described a syntactically restricted functional language which ensures statically fixed memory allocation and still surpasses primitive recursion. Their approach was focused on the use of their language for embedded systems heavily using parallel computation.

- Martin Hofmann [Hof00] constructed a translation for linearly typed functional code into C-code which preallocates all required heap-space, therefore rendering a Garbage Collection obsolete. Any heap-space used (and reused) must be given through a program's input; there are no limitations on the stack size. The complexity class covered by the functional language provably equals exponential time.

In each of the cited works the programmer would be required to care about the resource consumption of the programmed code by annotating resource related information, e.g. to the types. While it might be desirable in some applications of functional programming techniques to have an explicit control over resources, it is certainly not in general, because the concept of memory resources is naturally alien to the state-free philosophy of functional programming. Furthermore it seems that maintaining program code might also require drastic changes in the resource annotations. Therefore methods for an automatic inference of these resource annotations seem desirable, and most authors above already mention an extension of their presented approach in that way. Mads Tofte and Lars Birkedal [T&B98] already presented an algorithm implementing the specification of the Region Inference System cited above, which accepts unannotated input code.

We will now provide a system to infer the required resource annotations in the approach of Martin Hofmann. Let us therefore recall his work in more detail. As already said, his approach allows only the determination of the heap-space usage. Determining the overall space consumption of a functional program would also require to estimate the maximal stack-space usage. Calculating the maximal stack-space usage in turn equals to computing the maximal recursion-depth, which then amounts either to restrict to tail-recursion or to determine the time-usage of the program. We refrain from this part of the problem by solely concentrating on the heap-space usage of a given program, as the dynamic space allocation is the main difficulty to be overcome by a Garbage Collection.

The linear first-order functional programming language introduced by Martin Hofmann in [Hof00] was named LFPL. The language LFPL restricts manipulations of heap-allocated data structures like lists or trees to in-place modifications only. This is achieved by the use of a resource type denoted by $\Diamond$, which controls the number of constructors and destructors used: Each destructor additionally returns arguments of type $\Diamond$, whereas each constructor consumes a number of elements of type $\Diamond$. Although the elements of type $\Diamond$ are indistinguishable within the functional program itself, each represents a distinct small portion of available heap-space. Martin Hofmann then gives a translation of LFPL into malloc()-free C-code[3] and proves the correctness of this translation. He also shows that the expressive power of LFPL is precisely what can be computed on a linearly space bounded Turing machine having an unbounded stack as an extra resource, which equals the complexity class 'exponential time'.

So strictly speaking a LFPL-program automatically allows us to derive a rigid bound on the heap-space usage: namely zero, as it is impossible to allocate new heap-space within LFPL as the existence of the translation into malloc()-free C-code shows. Yet this is not true in a more open sense: By the same observation it follows that any initial data structures must be provided externally to an LFPL-program. Hence a LFPL-function may claim additional resources in form of additional arguments of type $\Diamond$. Furthermore dynamical space allocation is allowed in a controlled way in LFPL, for example by a list-processing function requiring that each list-node of the input-list additionally contains an element of type $\Diamond$. Thus the heap-space usage of a LFPL-program can be read off from its signature. Note that we deal with a slight variant of LFPL here, the changes and the complete details of LFPL are given in Appendix A.2.1.

Now our main goal is seeking relief from explicitly handling the resource type $\Diamond$ and hence we consider to what extent the usage of the resource type $\Diamond$ can be inferred automatically via static program analysis. A static program analysis is sufficient as all heap-space must be preallocated in LFPL. We therefore introduce the linearly typed language LF without a resource type like $\Diamond$ (or any other memory usage restricting mechanism) and ask ourselves how to translate LF into the language LFPL, thus deriving the heap-space usage of the LF-program from its translation. For convenience LF resembles LFPL except that it does not contain the resource type $\Diamond$, thus constructors simply do not need an argument of type $\Diamond$. See Definition A.3.1 for a formal definition of LF.

[3] In the language C new heap space is allocated by a call to the function malloc().

So given an LF-program P we want to compute its heap-space usage by finding an 'equivalent' LFPL-program P', where these bounds are clearly determinable from the signature of the program.

What do we mean by "an 'equivalent' program"? Well, intentionally any program that "does the same" *modulo* operations dealing with the resource type $\Diamond$ only. As extensional equality of programs is generally undecidable, we cannot hope to construct such an according LFPL-program whenever one exists in a feasible way. However this was not our aim, as we are merely interested in calculating the resource bounds of the concrete LF-program P and not of any optimised variants of P that may possibly exist.[4] So instead of defining abstract semantics for both languages – which would then require a justification for our purpose again – we define a map $|\cdot| : \mathsf{LFPL} \to \mathsf{LF}$ defining the modulo operation, i.e. that just *strips* all commands connected to the handling of the resource type $\Diamond$ from the program code. Then we can define our intended notion of equivalence by α-equivalence (i.e. modulo renaming of bound variables) of the image of the program code under $|\cdot|$. Hence we have a semantics of LFPL in terms of the language LF.

We then can reformulate our primary objective more precisely:

> *For all* LF*-programs* P*, construct a* LFPL*-program* P' *satisfying* $|P'| \stackrel{\alpha}{\equiv} P$*, whenever such a* P' *exists.*

The truth and meaning of such a statement apparently depends heavily upon the definition of $|\cdot|$, but we have a clear intention what this mapping should do and the definition of $|\cdot|$ can be done in a transparent way as we will show.[5]

In the following sections we will resolve the task of our primary objective by constructing *feasible* integer linear programs from a given LF-programs P and showing that it is possible to construct the LFPL-program P' if and only if the integer linear program (ILP) is solvable. We may sometimes refer to the names P and P' in this sense as just stated. The connected ILP will always be denoted by $(\!|P|\!)$.

We first reduce the problem of finding P' only within a fragment of the language LFPL; that fragment will be called $\mathsf{LFPL}_\Diamond$. In $\mathsf{LFPL}_\Diamond$ the explicit use of dynamical resources is prohibited, i.e. List- and Tree-types may not contain spare resources. $\mathsf{LFPL}_\Diamond$ is still capable of defining functions like Insertion-Sort or Quick-Sort, although its limits will be shown.

[4] Of course we are interested in methods of creating efficient programs in general, but within the scope of this work we are already contend by recognising programs with efficient memory usage.

[5] For an example of a natural translation from LFPL to LF, see Examples A.2.3 and A.3.3

In Section 2 we consider the full problem for LFPL as stated above. Overloading the symbols $(\!|\cdot|\!)$ and $|\cdot|$ for the different languages (or language-fragments) should not present a problem, as each section is more or less self-contained.

1 The simple case: $\mathsf{LFPL}_\Diamond$

This section shall demonstrate our general approach to the problem. For the sake of simplicity we restrict our search for P' to the following fragment of LFPL:

Definition 1.1 ($\mathsf{LFPL}_\Diamond$)**.** $\mathsf{LFPL}_\Diamond$ is a fragment of LFPL where $\Diamond$ is used only as a function parameter, i.e. the type grammar of $\mathsf{LFPL}_\Diamond$ is defined as follows:

$$\begin{aligned} &\text{zero-order types:} && P ::= \mathsf{1} \mid \mathsf{N} \mid \mathsf{L}(P) \mid \mathsf{T}(P) \mid P \otimes P \mid P + P \\ & && R ::= \Diamond \\ &\text{first-order types:} && F ::= (P, \ldots, P, R, \ldots, R) \to P \end{aligned}$$

The term grammar is identical to LFPL, though it is required that all occurring subterms have types according to this type grammar. Variables of type $\Diamond$ may therefore only occur in terms that explicitly require an argument of type $\Diamond$, e.g. function application terms and the construction and elimination terms for lists and trees. Adjusting the LFPL type rules for $\mathsf{LFPL}_\Diamond$ is then straightforwardly done by restricting the occurring type variables to types different from $\Diamond$, except where explicitly needed like in the type rules Variable and Function Application. For example the type rule Pairing for $\mathsf{LFPL}_\Diamond$ is

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : A_1 \qquad \Gamma_2 \vdash_\Sigma e_2 : A_2 \qquad A_1 \neq \Diamond \qquad A_2 \neq \Diamond}{\Gamma_1, \Gamma_2 \vdash_\Sigma e_1 \otimes e_2 : A_1 \otimes A_2}$$

while the $\mathsf{LFPL}_\Diamond$ type rule for List-Construction is

$$\frac{\Gamma_d \vdash_\Sigma e_d : \Diamond \qquad \Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A) \qquad A \neq \Diamond}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)}$$

and so the term e_d must eventually be a variable.

In order to construct $P' \in \mathsf{LFPL}_\Diamond$ according to a given LF-program P, we solely have to determine the number of arguments of type $\Diamond$ that are necessary to call a function $f_i \in P'$. Therefore we assign an integer variable x_i denoting this number to each f_i in the following way:

Definition 1.2 ($(\!|\cdot|\!) : \mathsf{LF} \to \mathrm{ILP}$)**.** For $P \in \mathsf{LF}$ containing exactly n different function symbols f_i, let $(\!|P|\!)$ denote the integer linear program over $x \in \mathbb{N}^n$ defined by

$$\min\left\{ \sum_{j=1}^{n} x_j \;\middle|\; \bigcup_{i=1}^{n} (\!|f_i(v_1, \ldots, v_k)|\!) \geq (\!|\text{defining body of } f_i(v_1, \ldots, v_k)|\!)\,,\ x \in \mathbb{N}^n \right\}$$

where the map

$$(\!|\cdot|\!) : \mathsf{LF\text{-}term} \longrightarrow (\mathbb{Z} \cup x, +, \max)$$

is recursively defined on the composition of e as shortly follows. $(\mathbb{Z} \cup x, +, \max) = \big(\mathbb{Z} \cup \bigcup_{i=1}^{n} x_i, +, \max\big)$ denotes the commutative monoid of arithmetic expressions over $\mathbb{Z}$ and the indices of x, exclusively built with the operations $+$ and max(), i.e. linear arithmetic terms containing x_i's with positive factors only, like $-3 + \max(x_1, 1 + x_2)$ or $3+4+x_6+x_4-9+x_6 = -2+x_4+2x_6$, but not terms like $1-x_2$ or $\max(0, 1-x_1)$ which contain the indices of x with a negative factor.

$$
\begin{aligned}
&(\!|v|\!) = 0 && \text{(Variable)}\\
&(\!|c|\!) = 0 \qquad \text{for } c = * \text{ or } c \text{ an integer constant} && \text{(Constant)}\\
&(\!|e_1 \star e_2|\!) = (\!|e_1|\!) + (\!|e_2|\!) \qquad \text{for } \star \in \{+, -, \times, \geq, \dots\} && \text{(Standard Integer Infix)}\\
&(\!|\mathsf{if}\ \ e_1\ \ \mathsf{then}\ \ e_2\ \ \mathsf{else}\ \ e_3|\!) = (\!|e_1|\!) + \max\big((\!|e_2|\!), (\!|e_3|\!)\big) && \text{(Conditional)}\\
&(\!|e_1 \otimes e_2|\!) = (\!|e_1|\!) + (\!|e_2|\!) && \text{(Pairing)}\\
&(\!|\mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2|\!) = (\!|e_1|\!) + (\!|e_2|\!) && \text{(Pair-Elimination)}\\
&(\!|\mathsf{inl}\,(e)|\!) = (\!|e|\!) && \text{(Left-Injection)}\\
&(\!|\mathsf{inr}(e)|\!) = (\!|e|\!) && \text{(Right-Injection)}\\
&(\!|\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{inl}(v) \Rightarrow e_2\ |\mathsf{inr}(v) \Rightarrow e_3|\!)\\
&\quad = (\!|e_1|\!) + \max\big((\!|e_2|\!), (\!|e_3|\!)\big) && \text{(Sum-Elimination)}\\
&(\!|\mathsf{nil}|\!) = 0 && \text{(Empty List)}\\
&(\!|\mathsf{cons}(e_1, e_2)|\!) = 1 + (\!|e_1|\!) + (\!|e_2|\!) && \text{(List-Construction)}\\
&(\!|\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{nil} \Rightarrow e_2\ \ |\mathsf{cons}(x_h, x_t) \Rightarrow e_3|\!)\\
&\quad = (\!|e_1|\!) + \max\big((\!|e_2|\!), (\!|e_3|\!) - 1\big) && \text{(List-Elimination)}\\
&(\!|\mathsf{leaf}|\!) = 0 && \text{(Leaf)}\\
&(\!|\mathsf{node}(e_a, e_l, e_r)|\!) = 1 + (\!|e_a|\!) + (\!|e_l|\!) + (\!|e_r|\!) && \text{(Tree-Node)}\\
&(\!|\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{leaf} \Rightarrow e_2\ |\mathsf{node}(x_a, x_l, x_r) \Rightarrow e_3|\!)\\
&\quad = (\!|e_1|\!) + \max\big((\!|e_2|\!), (\!|e_3|\!) - 1\big) && \text{(Tree-Elimination)}\\
&(\!|f_i(e_1, \dots, e_n)|\!) = x_i + (\!|e_1|\!) + \dots + (\!|e_n|\!) && \text{(Function Application)}
\end{aligned}
$$

Note that $x_i = (\!|f_i(v_1, \dots, v_{n_i})|\!)$ holds for $1 \leq i \leq n$.

Observation 1.3. Gordon Plotkin remarked that resources used to compute arithmetic expressions might be reused immediately, probably reducing the upper bound on resource consumption. Therefore we should define

$$(\!|e_1 \star e_2|\!) = \max\big((\!|e_1|\!), (\!|e_2|\!)\big) \qquad \text{for } \star \in \{+, -, \times, \geq, \dots\}$$

in the case of a Standard Integer Infix operation. It is not yet entirely clear if this principle may be further expanded to *all* base type expressions wherever they appear and without any complications arising, especially in a concrete implementation. We leave this question to future investigations.

$(\!|P|\!)$ as defined above is not exactly a linear program, since the inequalities contain the function max. In order to obtain a true integer linear program we have to successively replace each inequality of the kind

$$x_i \geq W_0 + \max(W_1, W_2)$$

by the two inequalities

$$\begin{aligned} x_i &\geq W_0 + W_1 \\ x_i &\geq W_0 + W_2 \end{aligned}$$

where W_i stands for an arbitrary expression in $(\mathbb{Z} \cup x, +, \max)$. Although this leads to an equivalent integer linear program, the number of inequalities may rise exponentially: If W_0 is of the form $\big(W_7 + \max(W_5, W_6)\big) + \max(W_3, W_4)$, i.e. contains further max-operations, we are forced to replace the two inequalities from above by

$$\begin{aligned} x_i &\geq W_7 + W_5 + W_3 + W_1 & \qquad x_i &\geq W_7 + W_5 + W_3 + W_2 \\ x_i &\geq W_7 + W_6 + W_3 + W_1 & \qquad x_i &\geq W_7 + W_6 + W_3 + W_2 \\ x_i &\geq W_7 + W_5 + W_4 + W_1 & \qquad x_i &\geq W_7 + W_5 + W_4 + W_2 \\ x_i &\geq W_7 + W_6 + W_4 + W_1 & \qquad x_i &\geq W_7 + W_6 + W_4 + W_2 \end{aligned}$$

This exponential blow up is caused by program constructs like

$$\mathsf{if}\ \big(\mathsf{if}\ (\mathsf{if}\ e_7\ \mathsf{then}\ e_5\ \mathsf{else}\ e_6)\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4\big)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$

e.g. when a branching operation is nested within the guard of another branching operation.[6] However it is possible to avoid this exponential blow up by introducing new variables; the equation

$$x_i \geq \Big(\big(W_7 + \max(W_5, W_6)\big) + \max(W_3, W_4)\Big) + \max(W_1, W_2)$$

considered above can equivalently be replaced by

$$\begin{aligned} x_i &\geq y + W_1 & \qquad y &\geq z + W_3 & \qquad z &\geq W_7 + W_5 \\ x_i &\geq y + W_2 & \qquad y &\geq z + W_4 & \qquad z &\geq W_7 + W_6 \end{aligned}$$

where $y, z \in \mathbb{N}$ are freshly introduced variables; this obviously allows construction within linear time depending on the input.

However, by the tutorial nature of this section and for simplicities sake, we prefer to keep the dimension of the integer linear program constant to n, hence accept a

[6] Branching operations are: Conditional, Sum-, List- and Tree-Elimination; see Appendix A.1

possible exponential blow up of the number of inequalities. It shall be mentioned that this mechanism is naturally included in the different approach of the problem in section 2: in this more general approach a polynomial construction time of $(\!|P|\!)$ can be guaranteed without introducing new variables, albeit there will be variables present which function in the same way. We will recall this again during the construction of $(\!|P|\!)$ in section 2.

So finally we have obtained $m \geq n$ inequalities of the form

$$x_{i(l)} \geq c_l + \sum_{j=1}^{n} a_{lj}\, x_j$$

for $1 \leq l \leq m$. Furthermore, the definition ensures $a_{lj} \in \mathbb{N}$ and $c_l \in \mathbb{Z}$. Collecting all of the above, $(\!|P|\!)$ is an integer linear program of form:

$$Bx \geq c + Ax$$

where $A \in \mathbb{N}^{m \times n}$, $x \in \mathbb{N}^n$, $c \in \mathbb{Z}^m$, $B \in \{0,1\}^{m \times n}$ with $b_{lj} = \begin{cases} 1 & \text{if } i(l) = j \\ 0 & \text{otherwise} \end{cases}$

Note that although m may be big, the matrix A can be considered sparse.

Theorem 1.4. *The integer linear program $(\!|P|\!)$ is solvable in polynomial time.*

Proof. It is well-known that solving an arbitrary integer linear program is an $\mathcal{NP}$-complete problem, e.g. see [Sch86], while relaxing to an linear program is known to be in $\mathcal{P}$. We claim that already $x \in \mathbb{N}^n$ holds for any optimal solution x obtained in the relaxed case.

So solve $(\!|P|\!) \cup \{x \geq 0\}$ for $x \in \mathbb{Q}^m$ in polynomial time using a standard algorithm, with $x \geq 0$ defined pointwise. Let x be an optimal solution. Observe that the equations in $(\!|P|\!)$ ensure that x is non-negative. Since A is a non-negative matrix we have

$$Bx \geq c + Ax \geq \lfloor c + Ax \rfloor \geq c + A\lfloor x \rfloor$$

where $\lfloor \cdot \rfloor : \mathbb{R}^+ \to \mathbb{N}$ stands for truncation. As A and c are integer so is $c + A\lfloor x \rfloor$ and we deduce

$$B\lfloor x \rfloor = \lfloor Bx \rfloor \geq \lfloor c + Ax \rfloor \geq c + A\lfloor x \rfloor$$

due to the property of B that each row is a unit-vector. Therefore $\lfloor x \rfloor$ is a better solution than x. As x is assumed to be optimal we conclude that $x = \lfloor x \rfloor$. □

A solution to $(\!|P|\!)$ shall be enough for us to construct P', but first we need to determine the meaning of equivalence of programs of the two languages LF and $\mathsf{LFPL}_\Diamond$. As already described in the introduction, we will do this by giving an embedding $|\cdot| : \mathsf{LFPL}_\Diamond \to \mathsf{LF}$. We define:

Definition 1.5 ($|\cdot| : \mathsf{LFPL}_\Diamond \to \mathsf{LF}$)**.**

$$
\begin{aligned}
&|v| = v && \text{(Variable)}\\
&|c| = c && \text{(Constant)}\\
&|e_1 \star e_2| = |e_1| \star |e_2| && \text{(Standard Integer Infix)}\\
&|\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3| = \mathsf{if}\ |e_1|\ \mathsf{then}\ |e_2|\ \mathsf{else}\ |e_3| && \text{(Conditional)}\\
&|e_1 \otimes e_2| = |e_1| \otimes |e_2| && \text{(Pairing)}\\
&|\mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2|\\
&\quad = \mathsf{match}\ |e_1|\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow |e_2| && \text{(Pair-Elimination)}\\
&|\mathsf{inl}\,(e)| = \mathsf{inl}\,(|e|) && \text{(Left-Injection)}\\
&|\mathsf{inr}(e)| = \mathsf{inr}(|e|) && \text{(Right-Injection)}\\
&|\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{inl}(v) \Rightarrow e_2\ |\mathsf{inr}(v) \Rightarrow e_3|\\
&\quad = \mathsf{match}\ |e_1|\ \mathsf{with}\ |\mathsf{inl}(v) \Rightarrow |e_2|\ |\mathsf{inr}(v) \Rightarrow |e_3| && \text{(Sum-Elimination)}\\
&|\mathsf{nil}| = \mathsf{nil} && \text{(Empty List)}\\
&|\mathsf{cons}(v_\Diamond, e_1, e_2)| = \mathsf{cons}(|e_1|, |e_2|) && \text{(List -Construction)}\\
&|\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{nil} \Rightarrow e_2\ |\mathsf{cons}(x_\Diamond, x_h, x_t) \Rightarrow e_3|\\
&\quad = \mathsf{match}\ |e|_1\ \mathsf{with}\ |\mathsf{nil} \Rightarrow |e_2|\ |\mathsf{cons}(x_h, x_t) \Rightarrow |e_3| && \text{(List-Elimination)}\\
&|\mathsf{leaf}| = \mathsf{leaf} && \text{(Leaf)}\\
&|\mathsf{node}(v_d, e_a, e_l, e_r)| = \mathsf{node}(|e_a|, |e_l|, |e_r|) && \text{(Tree-Node)}\\
&|\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{leaf} \Rightarrow e_2\ |\mathsf{node}(x_\Diamond, x_a, x_l, x_r) \Rightarrow e_3|\\
&\quad = \mathsf{match}\ |e_1|\ \mathsf{with}\ |\mathsf{leaf} \Rightarrow |e_2|\ |\mathsf{node}(x_a, x_l, x_r) \Rightarrow |e_3| && \text{(Tree-Elimination)}\\
&|f_i\,(e_1, \ldots, e_n)| = f_i\big(|e'_1|, \ldots, |e'_m|\big)\\
&\quad \text{where } e' \text{ is derived from } e \text{ by eliminating all } e_i : \Diamond && \text{(Function Application)}
\end{aligned}
$$

We extend the map $|\cdot|$ to types and (componentwise) to signatures accordingly:

$$
\begin{array}{lll}
|\mathsf{1}| = \mathsf{1} & |A \otimes B| = A \otimes B & |\mathsf{L}(A)| = \mathsf{L}(A)\\
|\mathsf{N}| = \mathsf{N} & |A + B| = A + B & |\mathsf{T}(A)| = \mathsf{T}(A)\\
|\Diamond| = \mathsf{1} & |(A_1, \ldots, A_n) \to C| = (A_1, \ldots, A_m) \to C &
\end{array}
$$

where $A_1, ..., A_m \neq \Diamond$ and $A_{m+1}, \ldots, A_n = \Diamond$ respectively for $0 \leq m \leq n$.

Now we can deal with the construction of the $\mathsf{LFPL}_\Diamond$ program P' when given a LF program P and a solution to $(\!|P|\!)$. The construction of the $\mathsf{LFPL}_\Diamond$ terms is given by the following lemma.

Lemma 1.6. *Let $P \in \mathsf{LF}$ with signature Σ and $\eta \in \mathbb{N}^n$ a solution of $(\!|P|\!)$. There exists a $\mathsf{LFPL}_\Diamond$ signature Σ' with $|\Sigma'| = \Sigma$ such that for all sub-terms e contained in P, having type A in context Γ, there exists a $\mathsf{LFPL}_\Diamond$ term e' satisfying*

$$\Gamma \overset{\mathsf{LF}}{\vdash_\Sigma} e : A \implies \Gamma, d_1 : \Diamond, \ldots, d_{(\!|e|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash_{\Sigma'}} e' : A \ \wedge \ |e'| \overset{\alpha}{\equiv} e$$

where $d_1, \ldots, d_{(\!|e|\!)_\eta}$ are assumed to be fresh variable names not occurring in term e. $(\!|e|\!)_\eta$ denotes the number derived by instantiating the variables of the $(\mathbb{Z} \cup x, +, \max)$-term $(\!|e|\!)$ according to η.

Proof. Construct Σ' accordingly to Σ by adding $(x_i)_\eta$ arguments of type $\Diamond$ to each function $f_i \in \mathrm{dom}\big(\Sigma\big)$, hence obviously $|\Sigma'| = \Sigma$ holds. The existence of e' then follows by induction on the composition of e. The LF and $\mathsf{LFPL}_\Diamond$ type rules differ only on List-/Tree-Construction, List-/Tree-Elimination and Function Application. So all other cases are trivial after the use of the induction hypothesis on the contained sub-terms of e and a proper renaming of the new introduced variables. Therefore we only exhibit some representative cases:

Variable: $\Gamma \overset{\mathsf{LF}}{\vdash_\Sigma} v : A$

By $(\!|v|\!) = 0$, $|v| = v$ and $\Gamma(v) = A$ we see that $e' = e$ is already the term we require:

$$\Gamma \overset{\mathsf{LFPL}_\Diamond}{\vdash_\Sigma} v : A$$

Conditional: $\Gamma_1, \Gamma_2 \overset{\mathsf{LF}}{\vdash_\Sigma} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A$

By the induction hypothesis we obtain the terms e'_1, e'_2, e'_3 satisfying

$$\Gamma_1, d_1 : \Diamond, \ldots, d_{(\!|e_1|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash_{\Sigma'}} e'_1 : \mathsf{N}$$

$$\Gamma_2, d_1 : \Diamond, \ldots, d_{(\!|e_2|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash_{\Sigma'}} e'_2 : A \quad \Gamma_2, d_1 : \Diamond, \ldots, d_{(\!|e_3|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash_{\Sigma'}} e'_3 : A$$

Since we want to construct the term e' out of these three sub-terms we need to rename the d_i. Let $\mu = \max\big((\!|e_2|\!)_\eta, (\!|e_3|\!)_\eta\big)$ then

$$\Gamma_2, d_{1+(\!|e_1|\!)} : \Diamond, \ldots, d_{\mu+(\!|e_1|\!)} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash_{\Sigma'}} e''_2 : A$$

$$\Gamma_2, d_{1+(\!|e_1|\!)} : \Diamond, \ldots, d_{\mu+(\!|e_1|\!)} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash_{\Sigma'}} e''_3 : A$$

where e''_2 is e'_2 after substituting d_i by $d_{i+(\!|e_1|\!)_\eta}$ for $i = 1, \ldots, (\!|e_2|\!)_\eta$ and similarly e''_3 being derived from e'_3 by substituting d_i by $d_{i+(\!|e_1|\!)_\eta}$ for $i = 1, \ldots, (\!|e_3|\!)_\eta$. Note that $|e'_2| \overset{\alpha}{\equiv} |e''_2|$ and $|e'_2| \overset{\alpha}{\equiv} |e''_3|$ as the $d_i : \Diamond$ are eliminated by $|\cdot|$. We conclude

$$\Gamma_1, \Gamma_2, d_1 : \Diamond, \ldots, d_{(\!|e_1|\!)_\eta+\mu} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash_{\Sigma'}} \mathsf{if}\ e'_1\ \mathsf{then}\ e''_2\ \mathsf{else}\ e''_3 : A$$

since by definition $(\!|\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3|\!)_\eta = (\!|e_1|\!)_\eta + \max\big((\!|e_2|\!)_\eta, (\!|e_3|\!)_\eta\big) = (\!|e_1|\!)_\eta + \mu$ and $|\mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'| \stackrel{\alpha}{\equiv} \mathsf{if}\ |e_1'|\ \mathsf{then}\ |e_2''|\ \mathsf{else}\ |e_3''|$.

Now let us deal with the slightly less obvious cases only:

List-Construction: $\Gamma_h, \Gamma_t \overset{\mathsf{LF}}{\vdash}_\Sigma \mathsf{cons}(e_h, e_t) : \mathsf{L}(A)$

By the induction hypothesis we have

$$\Gamma_h, d_1 : \Diamond, \ldots, d_{(\!|e_h|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash}_{\Sigma'} e_h'' : A$$
$$\Gamma_t, d_{(\!|e_h|\!)_\eta+1} : \Diamond, \ldots, d_{(\!|e_h|\!)_\eta+(\!|e_t|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash}_{\Sigma'} e_t'' : \mathsf{L}(A)$$

For the sake of simplicity we already renamed the d_i like shown in the previous case. By definition follows $(\!|\mathsf{cons}(e_h, e_t)|\!)_\eta = (\!|e_h|\!)_\eta + (\!|e_t|\!)_\eta + 1$, hence we obtain

$$\Gamma_h, \Gamma_t, d_1 : \Diamond, \ldots, d_{(\!|\mathsf{cons}(e_h,e_t)|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash}_{\Sigma'} \mathsf{cons}(d_{(\!|e_h|\!)_\eta+(\!|e_t|\!)_\eta+1}, e_h'', e_t'') : \mathsf{L}(A)$$

as $|\mathsf{cons}(d_{(\!|e_h|\!)_\eta+(\!|e_t|\!)_\eta+1}, e_h'', e_t'')| \stackrel{\alpha}{\equiv} \mathsf{cons}(|e_h''|, |e_t''|)$.

List-Elimination: $\Gamma_1, \Gamma_2 \overset{\mathsf{LF}}{\vdash}_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{nil} \Rightarrow e_2\ |\mathsf{cons}(h, t) \Rightarrow e_3 : C$

By appeal to the induction hypothesis and after renaming the d_i we obtain

$$\Gamma_1, d_1 : \Diamond, \ldots, d_{(\!|e_1|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash}_{\Sigma'} e_1' : \mathsf{L}(A)$$
$$\Gamma_2, d_{(\!|e_1|\!)_\eta+1} : \Diamond, \ldots, d_\mu : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash}_{\Sigma'} e_2' : C$$
$$\Gamma_2, h : A, t : \mathsf{L}(A)\,, d_{(\!|e_1|\!)_\eta+1} : \Diamond, \ldots, d_{\mu+1} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash}_{\Sigma'} e_3' : C$$

where $\mu = (\!|e_1|\!)_\eta + \max\big((\!|e_2|\!)_\eta, (\!|e_3|\!)_\eta - 1\big)$ and hence

$$(\!|e_2|\!)_\eta \leq \mu - (\!|e_1|\!)_\eta$$
$$(\!|e_3|\!)_\eta \leq \mu + 1 - (\!|e_1|\!)_\eta$$

So we exhibit as required

$$\Gamma_1, \Gamma_2, d_1 : \Diamond, \ldots, d_{(\!|e|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}}{\vdash}_\Sigma \mathsf{match}\ e_1'\ \mathsf{with}\ |\mathsf{nil} \Rightarrow e_2'\ |\mathsf{cons}(d_{\mu+1}, h, t) \Rightarrow e_3'$$

since

$$\left(\begin{array}{l}\mathsf{match}\ e_1'\ \mathsf{with}\ |\mathsf{nil} \Rightarrow e_2' \\ \qquad\qquad\quad |\mathsf{cons}(d_\mu, h, t) \Rightarrow e_3'\end{array}\right) \;=\; (\!|e_1|\!)_\eta + \max\big((\!|e_2|\!)_\eta, (\!|e_3|\!)_\eta - 1\big) = \mu$$

and

$$\left|\begin{array}{l}\mathsf{match}\ e_1'\ \mathsf{with}\ |\mathsf{nil} \Rightarrow e_2' \\ \qquad\qquad\qquad\quad |\mathsf{cons}(d_\mu, h, t) \Rightarrow e_3'\end{array}\right| \overset{\alpha}{\equiv} \begin{array}{l}\mathsf{match}\ |e_1'|\ \mathsf{with}\ |\mathsf{nil} \Rightarrow |e_2'| \\ \qquad\qquad\qquad\quad |\mathsf{cons}(h, t) \Rightarrow |e_3'|\end{array}$$

by definition.

Tree-Node, Tree-Elimination: The cases are similar to List-Construction and List-Elimination respectively, so we omit them here as well.

Function Application: $\Gamma_1, \ldots, \Gamma_m \overset{\mathsf{LF}}{\vdash}_\Sigma f_i(e_1, \ldots, e_m) : B$

We assume that $\Sigma(f_i) = (A_1, \ldots, A_m) \to B$ hence

$$\Sigma'(f_i) = (A_1, \ldots, A_m, \underbrace{\Diamond, \ldots, \Diamond}_{(x_i)_\eta}) \to B$$

By the induction hypothesis we obtain for $i = 1, \ldots, m$

$$\Gamma_i, d_1 : \Diamond, \ldots, d_{(\!|e_i|\!)_\eta} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash}_{\Sigma'} e_i' : A_i$$

We know that $(\!|f_i(e_1, \ldots, e_m)|\!)_\eta = (x_i)_\eta + (\!|e_1|\!)_\eta + \cdots + (\!|e_m|\!)_\eta$ by definition of $(\!|\cdot|\!)$, hence there are enough resources available to form the term

$$\Gamma_1, \ldots, \Gamma_m, d_1 : \Diamond, \ldots, d_{(\!|f_i(e_1,\ldots,e_m)|\!)} : \Diamond \overset{\mathsf{LFPL}_\Diamond}{\vdash}_{\Sigma'} f_i(e_1'', \ldots, e_m'', d_1, \ldots, d_{(x_i)_\eta}) : B$$

where e_i'' is derived from e_i' by a proper renaming of the d_i as usual. Using the definition of $|\cdot|$ we verify that

$$|f_i\big(e_1'', \ldots, e_m'', d_1, \ldots, d_{(x_i)_\eta}\big)| \overset{\alpha}{\equiv} f_i\big(|e_1''|, \ldots, |e_m''|\big)$$

as required to complete this case.

□

The main result of this section then follows quickly:

Theorem 1.7. *For all* LF*-programs* P *holds:*

$$\exists P' \in \mathsf{LFPL}_\Diamond . |P'| \overset{\alpha}{\equiv} P \iff \text{The ILP } (\!|P|\!) \text{ is solvable}$$

Proof. **"⇐":** Assume that $\eta \in \mathbb{N}^n$ is a solution of $(\!|P|\!)$. We construct P' by the use of Lemma 1.6 on the defining body e_{f_i} of each function $f_i \in P$, with context and type of the defining body according to the signature of f_i. Obviously $|P'| \overset{\alpha}{\equiv} P$ as Lemma 1.6 guarantees $|e_{f_i}| \overset{\alpha}{\equiv} e_{f_i}$ for each function.

"⇒": Let $P' \in \mathsf{LFPL}_\Diamond$ with $|P'| = P$. As $|P'| = P$ the programs can only differ on the explicit handling of resources. For each $f_i \in P'$ let

$$\eta_i = \text{number of arguments } A_i \text{ of type } \Diamond \text{ for } \Sigma'(f_i) = A_1, \ldots, A_m \to B$$

Now η must be a solution for $(\!|P|\!)$: By inspection of Definition 1.2 we see that $(\!|P|\!)$ consists of inequalities giving lower bounds on the number of resources consumed in each of the computational branches of each function $f_i \in P$. The program P' is assumed to be a valid $\mathsf{LFPL}_\Diamond$ program. Hence each function must have enough arguments of type $\Diamond$ to satisfy all the lower bounds on the resource consumption on each of its computational branches, as there is no other source of resources available in $\mathsf{LFPL}_\Diamond$.

□

Example 1.8 (Insertion-Sort)**.** As a short example we will now examine the well-known insertion-sort algorithm. Let $P_{IS}, \Sigma_{IS} \in \mathsf{LF}$ be as follows:

$$\begin{aligned}
\Sigma_{IS}(\mathtt{sort}) &= (\mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N}) \\
\Sigma_{IS}(\mathtt{ins}) &= (\mathsf{N}, \mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N})
\end{aligned}$$

$$\begin{aligned}
\mathtt{ins}(n, l) = \mathsf{match}\ l\ \mathsf{with}\ &\mid \mathsf{nil} \Rightarrow \mathsf{cons}(n, \mathsf{nil}) \\
&\mid \mathsf{cons}(h, t) \Rightarrow \mathsf{if}\ n \leq h\ \mathsf{then}\ \mathsf{cons}(n, \mathsf{cons}(h, t)) \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{cons}(h, \mathtt{ins}(n, t)) \\
\mathtt{sort}(l) = \mathsf{match}\ l\ \mathsf{with}\ &\mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
&\mid \mathsf{cons}(h, t) \Rightarrow \mathtt{ins}(h, \mathtt{sort}(t))
\end{aligned}$$

Note that the presented program code is strictly speaking not written in a linear style, as the variables n and h are used twice in the definition of `ins`: once in the

guard of the conditional and once in the branches. Of course, variables may always be shared between computational branches without violating linear typing, but in this case the variables are used in the guard as well. For the moment just note that we may relax linear typing to allow the multiple use of variables of base types like N and $\mathsf{1}$ (but of course not $\Diamond$) without any problems arising by adding a certain type rule. We will deal with this in detail in Section 3.1, but for simplicity we already allow us to make use of this rule in the given examples yet without further notice.

Computing $(\!|P_{IS}|\!)$ then results in the integer linear program

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{\mathtt{ins}} \\ x_{\mathtt{sort}} \end{pmatrix} \geq \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ -1 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_{\mathtt{ins}} \\ x_{\mathtt{sort}} \end{pmatrix} \qquad (x \in \mathbb{N}^2)$$

which can be transformed to the equivalent system

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} x_{\mathtt{ins}} \\ x_{\mathtt{sort}} \end{pmatrix} \geq \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} \qquad (x \in \mathbb{N}^2)$$

having the (unique) optimal solution $x = \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right)$. This leads us straightforwardly to the LFPL-program as stated below

$$\begin{aligned} \Sigma'_{IS}(\mathtt{sort}') &= (\mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N}) \\ \Sigma'_{IS}(\mathtt{ins}') &= (\mathsf{N}, \mathsf{L}(\mathsf{N}), \Diamond) \to \mathsf{L}(\mathsf{N}) \end{aligned}$$

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

Another program example that this technique already covers is the Quicksort algorithm as presented later in Example 3.1.3. Note that dynamical resource allocation is prohibited in $\mathsf{LFPL}_{\Diamond}$. As an example for this we consider the simple function `twice`, which just doubles each entry of an integer list:

Example 1.9 (Twice)**.**

$$\begin{aligned}
&\texttt{twice} : (\mathsf{L}(N)) \to \mathsf{L}(N)\\
&\texttt{twice}(l) = \mathsf{match}\ l\ \mathsf{with}\ |\mathsf{nil} \Rightarrow \mathsf{nil}\\
&\qquad\qquad\qquad |\mathsf{cons}(h,t) \Rightarrow \mathsf{cons}\big(h, \mathsf{cons}(h, \texttt{twice}(t))\big)
\end{aligned}$$

Computing $(\!|\texttt{twice}(v)|\!)$ yields the inequality

$$x_{\texttt{twice}} \geq 0 + \max\big\{0,\, -1 + 1 + 1 + x_{\texttt{twice}}\big\} = \max\big\{0,\, 1 + x_{\texttt{twice}}\big\}$$

which is apparently not solvable. The problem is that the number of resources required to call function `twice` dynamically depends on the length of the input list: For each list-node we need an additional resource to place its copy within. (The problem has nothing to do with the double appearance of variable h, violating a linear typing. As already mentioned above, copying elements of N can be done without problems. We could easily restore strict linearity by replacing the second appearance of h by a constant number, without changing the crucial part of this example.)

The methods provided in the next section will be capable of handling functions like `twice`, which require dynamical allocation of resources depending on their input. We will return to function `twice` in Example 3.2.1.

2 The complex case: LFPL and $\widehat{\mathsf{LFPL}}$

Now we want to expand our search for translations of LF-programs P to the full range of programs $P' \in \mathsf{LFPL}$. Let us first discuss some approaches of defining $|\cdot|$ in a natural way by the example of a cons operation on the List-type, as this already turns out to be problematic:

2.1 Naturally defining $|\cdot| : \mathsf{LFPL} \to \mathsf{LF}$

- The naive approach as in Definition 1.5: $|\mathsf{cons}(e_d, e_h, e_t)| = \mathsf{cons}(|e_h|, |e_t|)$
  Thus simply eliminating the term $e_d : \Diamond$ which provides the necessary resource needed to construct a list-node. Now $|\cdot|$ would not preserve the standard semantics of a program as the evaluation of e_d could be non-terminating, e.g.

$$e_d = \mathtt{borr}(*) \qquad \text{where} \quad \begin{array}{l} \mathtt{borr} : 1 \to \Diamond \\ \mathtt{borr}(x) := \mathtt{borr}(x) \end{array}$$

  So we could not know whether there exists at least one terminating LFPL program P' with $|P'| \overset{\alpha}{\equiv} P$ or not. This would be unsatisfying as we can always trivially construct a LFPL program P' with $|P'| \overset{\alpha}{\equiv} P$ that, whenever a resource is needed, just 'borrows' this resource by a call to the function borr as defined above at the cost of non-termination.

  Note that this was not an issue in Section 1, as the only terms of type $\Diamond$ allowed in $\mathsf{LFPL}_\Diamond$ are variables.

- $|\mathsf{cons}(e_d, e_h, e_t)| = |e_d|; \mathsf{cons}(|e_h|, |e_t|)$
  This would be an unpleasant solution as well, as there are completely unmotivated artifacts of the resource type handling remaining within the LF-program. So for a function f like

$$f : \mathsf{N}, \mathsf{L}(\mathsf{N}) \to \mathsf{L}(\mathsf{N}) \qquad f(x, l) := \mathsf{cons}(x, l)$$

  it might be true that $\forall f' \in \mathsf{LFPL}.|f'| \overset{\alpha}{\not\equiv} f$ although the function g

$$g : \mathsf{N}, \mathsf{L}(\mathsf{N}), \Diamond \to \mathsf{L}(\mathsf{N}) \qquad g(x, l, d) := \mathsf{cons}(d, x, l)$$

  would naturally be viewed as the LFPL-counterpart of f. After all we are seeking a LFPL-program according to an LF-program and not vice versa.

- By a similar argument we are led to abandon the approach of defining $|\cdot|$ to replace each occurrence of type $\Diamond$ by the unit type $\mathsf{1}$. A definition like this would already restrict the possible heap-space usage of P' within the LF-program P, as only the occurrence of a unit type argument would allow the use of one resource.

The solution to these problems lies within the following observations, which – in order to state them – require some simple definitions in advance. We will stick to the naive first approach to $|\cdot|$, which will turn out to be quite sensible under a mild assumption on the signature.

Definition 2.1.1. Define

$$\langle\cdot\rangle : \mathsf{LFPL}\text{-zero-order types} \longrightarrow \mathbb{N}$$

and say that $\langle A\rangle$ is the *content* of an LFPL-type A. The intuition of $\langle A\rangle$ is the number of resources that are at least contained in any element of type A. The defining equations are:

$$\begin{aligned} \langle\mathsf{1}\rangle &= 0 & \langle\mathsf{N}\rangle &= 0 & \langle\Diamond\rangle &= 1 \\ \langle\mathsf{L}(A)\rangle &= 0 & \langle\mathsf{T}(A)\rangle &= 0 & & \\ \langle A \otimes B\rangle &= \langle A\rangle + \langle B\rangle & \langle A + B\rangle &= \min\big(\langle A\rangle, \langle B\rangle\big) & & \end{aligned}$$

We also extend this map on arbitrary contexts to be the sum of the content of the type of each variable within the domain of the context:

$$\langle\Gamma\rangle = \sum_{x\in\mathrm{dom}(\Gamma)} \langle\Gamma(x)\rangle$$

Definition 2.1.2 (Faithful LFPL-signatures)**.** Any LFPL-signature Σ containing no first-order type with a result type of content greater than zero, i.e.

$$\forall f \in \mathrm{dom}(\Sigma).\, \Sigma(f) = (A_1, \ldots, A_n) \to B \implies \langle B\rangle = 0$$

is called a *faithful*

Lemma 2.1.3. *Let Σ be a faithful LFPL-signature, Γ a LFPL-typing context. Furthermore let e be a LFPL-term and A be a LFPL-type then the following holds:*

$$\Gamma \overset{\mathsf{LFPL}}{\vdash_\Sigma} e : A \implies \langle\Gamma\rangle \geq \langle A\rangle$$

Proof. By induction on the composition of term e:

Variable: $\Gamma \vdash_\Sigma v : \Gamma(v)$
By Definition 2.1.1 it follows immediately that $\langle\Gamma\rangle = \sum_{x\in\mathrm{dom}(\Gamma)} \Gamma(x) \geq \Gamma(v)$

Constant I+II, Integer Infix: $\Gamma \vdash_\Sigma e : A$
Trivial, as $\langle \mathsf{N}\rangle = 0$ and $\langle \mathsf{1}\rangle = 0$ in all these cases.

Conditional: $\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : C$
By the LFPL-typing rules and the induction hypothesis it follows $\langle\Gamma_2\rangle \geq \langle A\rangle$, hence $\langle\Gamma_1, \Gamma_2\rangle = \langle\Gamma_1\rangle + \langle\Gamma_2\rangle \geq \langle A\rangle$ as needed.

Pairing: $\Gamma_1, \Gamma_2 \vdash_\Sigma e_1 \otimes e_2 : A_1 \otimes A_2$
By the typing rules and the induction hypothesis we have: $\langle\Gamma_1\rangle \geq \langle A_1\rangle$ and $\langle\Gamma_2\rangle \geq \langle A_2\rangle$ therefore we conclude $\langle\Gamma_1, \Gamma_2\rangle \geq \langle A_1\rangle + \langle A_2\rangle = \langle A_1 \otimes A_2\rangle$.

Pair-Elimination: $\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2 : C$
Again by appeal to the typing rules and the induction hypothesis we obtain: $\langle\Gamma_1\rangle \geq \langle A_1 \otimes A_2\rangle = \langle A_1\rangle + \langle A_2\rangle$ and $\langle\Gamma_2\rangle + \langle A_1\rangle + \langle A_2\rangle \geq \langle C\rangle$ thus we derive that $\langle\Gamma_1, \Gamma_2\rangle \geq \langle C\rangle$.

Inl, Inr: $\Gamma \vdash_\Sigma \mathsf{inl}(e) : A + B$
By the typing rules and the induction hypothesis we derive

$$\langle\Gamma\rangle \geq \langle A\rangle \geq \min\big\{\langle A\rangle, \langle B\rangle\big\} = \langle A + B\rangle$$

The case of $\mathsf{inr}(\cdot)$ is similar.

Sum-Elimination: $\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{inl}(v) \Rightarrow e_2\ |\mathsf{inr}(v) \Rightarrow e_3 : C$
By typing rules and induction hypothesis we obtain

$$\langle\Gamma_1\rangle \geq \langle A + B\rangle = \min\big\{\langle A\rangle, \langle B\rangle\big\} \quad \langle\Gamma_2\rangle + \langle A\rangle \geq \langle C\rangle \quad \langle\Gamma_2\rangle + \langle B\rangle \geq \langle C\rangle$$

Thus either $\langle\Gamma_1\rangle \geq \langle A\rangle$ or $\langle\Gamma_1\rangle \geq \langle B\rangle$ and in both cases $\langle\Gamma_1, \Gamma_2\rangle \geq \langle C\rangle$ as needed.

Nil, List-Construction: $\Gamma \vdash_\Sigma e : \mathsf{L}(A)$
Trivial, as $\langle \mathsf{L}(A)\rangle = 0$ in both cases.

List-Elimination: $\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{nil} \Rightarrow e_2\ |\mathsf{cons}(d, h, t) \Rightarrow e_3 : C$
By typing rules and induction hypothesis applied to the nil-branch we obtain immediately that $\langle\Gamma_1, \Gamma_2\rangle \geq \langle\Gamma_2\rangle \geq \langle C\rangle$.

Leaf, Tree-Node: $\Gamma \vdash_\Sigma e : \mathsf{T}(A)$
Trivial, as $\langle \mathsf{T}(A) \rangle = 0$ in both cases.

Tree-Elimination: $\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{leaf} \Rightarrow e_2\ |\mathsf{node}(d, a, l, r) \Rightarrow e_3 : C$
By typing rules and induction hypothesis we obtain $\langle \Gamma_2 \rangle \geq \langle C \rangle$ from the leaf-branch. Hence we conclude $\langle \Gamma_1, \Gamma_2 \rangle \geq \langle C \rangle$.

Function Application: $\Gamma_1, \ldots, \Gamma_n \vdash_\Sigma f(e_1, \ldots, e_n) : B$
Trivial, as $\langle B \rangle = 0$ by assumption.

□

Let us look at our example of the naive approach to $|\cdot|$ again:

$$|\mathsf{cons}(e_d, e_h, e_t)| = \mathsf{cons}(|e_h|, |e_t|)$$

as $e_d : \Diamond$, the term e_d cannot be of importance for the result of the computation, since all elements of type $\Diamond$ are indistinguishable and there are no side-effects in a pure functional language except for non-termination due to recursive function calls. By Lemma 2.1.3 we know that the needed resource must already be available, as there are no resource allocation operators in LFPL, so all the term e_d eventually can do is separating this resource from compounds already contained in the context. So by restricting the signature to a faithful one we know that operations like function calls are always unnecessary in expressions of type $\Diamond$. We will formalise this observation by defining a new language, where the described problem of defining $|\cdot|$ in the naive way naturally does not arise.

Before we go on, please note that the imposed restrictions due to faithful signatures are not at all completely natural: one might be interested in programming functions like

$$\mathtt{head} : \mathsf{L}(A) \to (A \otimes \Diamond) \otimes \mathsf{L}(A) \qquad \begin{array}{l} \mathtt{head}(l) := \mathsf{match}\ l\ \mathsf{with} \\ \qquad |\mathsf{nil} \quad\quad\quad\quad\ \Rightarrow \mathtt{head}(\mathsf{nil}) \\ \qquad |\mathsf{cons}(d, h, t) \quad \Rightarrow (h \otimes d) \otimes t \end{array}$$

which is intended to be used only on non-empty lists. In some cases additional reasoning may allow save and convenient use of such constructs, but our restriction to faithful signatures prohibits programming styles like these. So a function like `head` must be properly implemented like

$$\mathtt{head} : \mathsf{L}(A) \to \mathsf{1} + \big((A \otimes \Diamond) \otimes \mathsf{L}(A)\big) \quad \begin{array}{l} \mathtt{head}(l) := \mathsf{match}\ l\ \mathsf{with} \\ \qquad |\mathsf{nil} \quad\quad\quad\quad\ \Rightarrow \mathsf{inl}(*) \\ \qquad |\mathsf{cons}(d, h, t) \quad \Rightarrow \mathsf{inr}\big((h \otimes d) \otimes t\big) \end{array}$$

when a faithful signature is required. It is not clear to us whether the restriction to faithful signatures restricts expressive power, but it seems unlikely.

2.2 The language $\widehat{\mathsf{LFPL}}$

We introduce a restricted variant of LFPL now, named $\widehat{\mathsf{LFPL}}$, which allows us only to build programs with a faithful signature and that also enforces a 'normalised' handling of the resource type $\Diamond$, justified by what we have just seen. We will then discuss the strong connection between LFPL and $\widehat{\mathsf{LFPL}}$ and prove equality of expressive power when restricting LFPL to programs with faithful signatures.

Definition 2.2.1 ($\widehat{\mathsf{LFPL}}$)**.** We will define the type and term grammar of $\widehat{\mathsf{LFPL}}$ and the typing-rules now. The type grammar:

$$\begin{aligned}
\text{pure zero-order types:} \quad & P ::= \mathsf{1} \mid \mathsf{N} \mid \mathsf{L}(R) \mid \mathsf{T}(R) \mid P \otimes P \mid P + R \mid R + P \\
\text{rich zero-order types:} \quad & R ::= (P, n) \quad \text{where } n \in \mathbb{N} \\
\text{(pure) first-order types:} \quad & F ::= (P, \ldots, P, n) \to P \quad \text{where } n \in \mathbb{N}
\end{aligned}$$

To provide a glimpse where this will lead us, the $\widehat{\mathsf{LFPL}}$-type $(N, 2)$ is set to correspond to the LFPL-type $N \otimes (\Diamond \otimes \Diamond)$ as well as with all product permutations of this type. So we extend Definition 2.1.1 to $\widehat{\mathsf{LFPL}}$ by:

$$\langle (A, n) \rangle = n \qquad\qquad \langle B \rangle = 0$$

where (A, n) is an arbitrary rich $\widehat{\mathsf{LFPL}}$ zero-order type and B is any pure $\widehat{\mathsf{LFPL}}$ zero-order type. We therefore may occasionally allow ourselves to conveniently use the pure type B instead of the rich type $(B, 0)$, but never vice versa.

Note that it is not possible to include a single sum-type $R + R$, as the content of such a type could be non-zero, hence functions returning sum-types could possibly violate our desired restriction on the signature. So LFPL-types like $(A \otimes \Diamond) + \big((B \otimes \Diamond) \otimes \Diamond\big)$ must correspond to the rich $\widehat{\mathsf{LFPL}}$-type $\big(A + (B, 1), 1\big)$.

The terms of $\widehat{\mathsf{LFPL}}$ are given by the following grammar:[7]

$e ::=$	v	Variable
	$\mid c$	Constant
	$\mid e_1 \star e_2$	Standard Integer Infix
	$\mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$	Conditional
	$\mid e_1 \otimes e_2$	Pairing
	$\mid \mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2$	Pair-Elimination
	$\mid \mathsf{inl}(e)$	Left Injection
	$\mid \mathsf{inr}(e)$	Right Injection
	$\mid \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{inl}(v) \Rightarrow e_2$	Sum-Elimination
	$\qquad\qquad\qquad\quad \mid \mathsf{inr}(v) \Rightarrow e_3$	
	$\mid \mathsf{nil}$	Empty List
	$\mid \mathsf{cons}(e_h, e_t)$	List-Construction
	$\mid \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_2$	List-Elimination
	$\qquad\qquad\qquad\quad \mid \mathsf{cons}(h, t) \Rightarrow e_3$	
	$\mid \mathsf{leaf}$	Leaf
	$\mid \mathsf{node}(e_a, e_l, e_r)$	Tree-Node
	$\mid \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{leaf} \Rightarrow e_2$	Tree-Elimination
	$\qquad\qquad\qquad\quad \mid \mathsf{node}(a, l, r) \Rightarrow e_3$	
	$\mid f(e_1, \ldots, e_n)$	Function Application

Let Σ be an $\widehat{\mathsf{LFPL}}$ signature mapping a finite set of function identifiers to $\widehat{\mathsf{LFPL}}$ first-order types, Γ be an $\widehat{\mathsf{LFPL}}$ typing context mapping a finite set of identifiers to $\widehat{\mathsf{LFPL}}$ pure zero-order types, and n a natural number. An $\widehat{\mathsf{LFPL}}$ typing judgement $\Gamma, n \vdash_\Sigma e : A$ then reads "the $\widehat{\mathsf{LFPL}}$ term e has type A under signature Σ and in typing context Γ with n free memory resources available." If the domains of Γ_1 and Γ_2 are disjoint we allow ourselves to write "$\Gamma_1, \Gamma_2, n \vdash_\Sigma e : A$" meaning that the typing context of this judgement consists of the disjoint union of Γ_1 and Γ_2. The natural number assorted with the context, the natural numbers occurring within the types and in the side conditions of the following type rules are sometimes referred to as *resource numbers*.

In each of the following type rules, let furthermore A, B, C denote arbitrary $\widehat{\mathsf{LFPL}}$ pure zero-order types and $n, k, m_1, m_2, \ldots \in \mathbb{N}$ denote arbitrary values. Then we

[7]Note that the term-grammar is identical to the term-grammar of LF given in Definition A.3.1!

allow the following typing judgements for $\widehat{\mathsf{LFPL}}$, provided that the premises above, the rule and all the equations noted at the side of each rule are satisfied.[8]

Variable

$$\frac{}{\Gamma, n \vdash_\Sigma v : \Gamma(v)}\ (v \in \mathrm{dom}(\Gamma))$$

Constant I+II

$$\frac{}{\Gamma, n \vdash_\Sigma * : \mathsf{1}} \qquad \frac{}{\Gamma, n \vdash_\Sigma c : \mathsf{N}}\ (c \text{ is a integer constant})$$

Standard Integer Infix Operator

$$\frac{\Gamma_1, m_1 \vdash_\Sigma e_1 : \mathsf{N} \qquad \Gamma_2, m_2 \vdash_\Sigma e_2 : \mathsf{N}}{\Gamma_1, \Gamma_2, n \vdash_\Sigma e_1 \star e_2 : \mathsf{N}} \begin{pmatrix} n \geq m_1 + m_2 \\ \text{and } \star \text{ is a integer} \\ \text{infix operator} \end{pmatrix}$$

Conditional

$$\frac{\begin{array}{c}\Gamma_2, m_2 \vdash_\Sigma e_2 : A \\ \Gamma_1, m_1 \vdash_\Sigma e_1 : \mathsf{N} \qquad \Gamma_2, m_3 \vdash_\Sigma e_3 : A\end{array}}{\Gamma_1, \Gamma_2, n \vdash_\Sigma \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A} \begin{pmatrix} n \geq m_1 + m_2 \\ n \geq m_1 + m_3 \end{pmatrix}$$

Pairing

$$\frac{\Gamma_1, m_1 \vdash_\Sigma e_1 : A_1 \qquad \Gamma_2, m_2 \vdash_\Sigma e_2 : A_2}{\Gamma_1, \Gamma_2, n \vdash_\Sigma e_1 \otimes e_2 : A_1 \otimes A_2}\ (n \geq m_1 + m_2)$$

Pair-Elimination

$$\frac{\begin{array}{c}\Gamma_1, m_1 \vdash_\Sigma e_1 : A_1 \otimes A_2 \\ \Gamma_2, v_1 : A_1, v_2 : A_2, m_2 \vdash_\Sigma e_2 : C\end{array}}{\Gamma_1, \Gamma_2, n \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2 : C}\ (n \geq m_1 + m_2)$$

Inl I

$$\frac{\Gamma, m \vdash_\Sigma e : A}{\Gamma, n \vdash_\Sigma \mathsf{inl}(e) : A + (B, k)}\ (n \geq m)$$

Inr I

$$\frac{\Gamma, m \vdash_\Sigma e : B}{\Gamma, n \vdash_\Sigma \mathsf{inr}(e) : A + (B, k)}\ (n \geq m + k)$$

[8]Note that the typing-rules of LF are essentially equal to those presented now, except for the added linear constraints.

Inl II

$$\frac{\Gamma, m \vdash_\Sigma e : A}{\Gamma, n \vdash_\Sigma \mathsf{inl}(e) : (A, k) + B} \, (n \geq m + k)$$

Inr II

$$\frac{\Gamma, m \vdash_\Sigma e : B}{\Gamma, n \vdash_\Sigma \mathsf{inr}(e) : (A, k) + B} \, (n \geq m)$$

Sum-Elimination I

$$\frac{\begin{array}{c}\Gamma_1, m_1 \vdash_\Sigma e_1 : A + (B, k) \\ \Gamma_2, v : A, m_2 \vdash_\Sigma e_2 : C \\ \Gamma_2, v : B, m_3 \vdash_\Sigma e_3 : C\end{array}}{\begin{array}{c}\Gamma_1, \Gamma_2, n \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{inl}\,(v) \Rightarrow e_2 : C \\ \mid \mathsf{inr}(v) \Rightarrow e_3\end{array}} \begin{pmatrix} n \geq m_1 + m_2 \\ n \geq m_1 + m_3 - k \end{pmatrix}$$

Sum-Elimination II

$$\frac{\begin{array}{c}\Gamma_1, m_1 \vdash_\Sigma e_1 : (A, k) + B \\ \Gamma_2, v : A, m_2 \vdash_\Sigma e_2 : C \\ \Gamma_2, v : B, m_3 \vdash_\Sigma e_3 : C\end{array}}{\begin{array}{c}\Gamma_1, \Gamma_2, n \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{inl}\,(v) \Rightarrow e_2 : C \\ \mid \mathsf{inr}(v) \Rightarrow e_3\end{array}} \begin{pmatrix} n \geq m_1 + m_2 - k \\ n \geq m_1 + m_3 \end{pmatrix}$$

Empty List

$$\frac{}{\Gamma, n \vdash_\Sigma \mathsf{nil} : \mathsf{L}(A, k)}$$

List-Construction

$$\frac{\Gamma_1, m_1 \vdash_\Sigma e_h : A \qquad \Gamma_2, m_2 \vdash_\Sigma e_t : \mathsf{L}(A, k)}{\Gamma_1, \Gamma_2, n \vdash_\Sigma \mathsf{cons}(e_h, e_t) : \mathsf{L}(A, k)} \, (n \geq m_1 + m_2 + (1 + k))$$

List-Elimination

$$\frac{\begin{array}{c}\Gamma_1, m_1 \vdash_\Sigma e_1 : \mathsf{L}(A, k) \\ \Gamma_2, m_2 \vdash_\Sigma e_2 : C \\ \Gamma_2, h : A, t : \mathsf{L}(A, k)\,, m_3 \vdash_\Sigma e_3 : C\end{array}}{\begin{array}{c}\Gamma_1, \Gamma_2, n \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_2 \qquad\qquad : C \\ \mid \mathsf{cons}(h, t) \Rightarrow e_3\end{array}} \begin{pmatrix} n \geq m_1 + m_2 \\ n \geq m_1 + m_3 - (1 + k) \end{pmatrix}$$

Leaf

$$\frac{}{\Gamma, n \vdash_\Sigma \mathsf{leaf} : \mathsf{T}(A,k)}$$

Tree-Node

$$\frac{\begin{array}{cc} & \Gamma_2, m_2 \vdash_\Sigma e_2 : \mathsf{T}(A,k) \\ \Gamma_1, m_1 \vdash_\Sigma e_1 : A & \Gamma_3, m_3 \vdash_\Sigma e_3 : \mathsf{T}(A,k) \end{array}}{\Gamma_1, \Gamma_2, \Gamma_3, n \vdash_\Sigma \mathsf{node}(e_1, e_2, e_3) : \mathsf{T}(A,k)} \, (n \geq (1+k) + \sum m_i)$$

Tree-Elimination

$$\frac{\begin{array}{c} \Gamma_1, m_1 \vdash_\Sigma e_1 : \mathsf{T}(A,k) \qquad \Gamma_2, m_2 \vdash_\Sigma e_2 : C \\ \Gamma_2, a : A, l : \mathsf{T}(A,k), r : \mathsf{T}(A,k), m_3 \vdash_\Sigma e_3 : C \end{array}}{\begin{array}{ll} \Gamma_1, \Gamma_2, n \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with} & \mid \mathsf{leaf} \Rightarrow e_2 \\ & \mid \mathsf{node}(a,l,r) \Rightarrow e_3 \end{array} : C} \begin{pmatrix} n \geq m_1 + m_2 \\ n \geq m_1 + m_3 - (1+k) \end{pmatrix}$$

Function Application

$$\frac{\begin{array}{c} \Sigma(f) = (A_1, \ldots, A_p, k) \longrightarrow B \\ \Gamma_i, m_i \vdash_\Sigma e_i : A_i \quad (i = 1, \ldots, p) \end{array}}{\Gamma_1, \ldots, \Gamma_p, n \vdash_\Sigma f(e_1, \ldots, e_p) : B} \, (n \geq k + \sum m_i)$$

Lemma 2.2.2. *Observe that the following rule*

Waste
$$\frac{\Gamma, m \vdash_\Sigma e : A}{\Gamma, n \vdash_\Sigma e : A} \, (n \geq m)$$

is admissible.

Proof. By inspection of all terminating $\widehat{\mathsf{LFPL}}$ type rules, i.e. those rules which do not require a typing judgement as a precondition. Due to the fact that all terminating type rules impose no restriction on the natural number counting the available resources, unused memory resources may be passed on to those. □

Set-Theoretic Semantics of $\widehat{\mathsf{LFPL}}$

The set-theoretic semantics of $\widehat{\mathsf{LFPL}}$ is simply an extension of the set-theoretic semantics of LF. We must solely add the interpretation of rich zero-order types.

Definition 2.2.3. The set-theoretic semantics of $\widehat{\mathsf{LFPL}}$ are identical to the the set-theoretic semantics of LF, given in Definition A.3.2, extended by

$$[\![(P, n)]\!] = [\![P]\!]$$

The set-theoretic semantics of $\widehat{\mathsf{LFPL}}$ shall only provide us with the formal denotation of an $\widehat{\mathsf{LFPL}}$ program. So again we ignore resource related concerns in the semantics as we did in Definition A.2.2, the set-theoretic semantics of LFPL.

The signature is indeed all that distinguishes an $\widehat{\mathsf{LFPL}}$-program from an LF-program. However, we still can guarantee that a $\widehat{\mathsf{LFPL}}$-program does not need to allocate any additional dynamic space by the close connection to LFPL.

2.3 The strong connection between LFPL and $\widehat{\mathsf{LFPL}}$

In order to prove the equality of the expressive power of $\widehat{\mathsf{LFPL}}$ and LFPL, we will show how to translate a given $\widehat{\mathsf{LFPL}}$-program into a corresponding[9] $\widehat{\mathsf{LFPL}}$-program and vice versa.

Transforming $\widehat{\mathsf{LFPL}}$ to LFPL

It is easy to see that any $\widehat{\mathsf{LFPL}}$-program can be transformed into a corresponding LFPL-program. Replace each $\widehat{\mathsf{LFPL}}$ type according to the following map.

Definition 2.3.1. Define the map $\phi : \widehat{\mathsf{LFPL}}$ types $\longrightarrow \mathsf{LFPL}$ types recursively as follows:

$$
\begin{array}{lll}
\mathsf{1} \longmapsto \mathsf{1} & \mathsf{L}(A) \longmapsto \mathsf{L}(\phi(A)) & A \otimes B \longmapsto \phi(A) \otimes \phi(B) \\
\mathsf{N} \longmapsto \mathsf{N} & \mathsf{T}(A) \longmapsto \mathsf{T}(\phi(A)) & A + B \longmapsto \phi(A) + \phi(B) \\
 & (A, r) \longmapsto \phi(A) \otimes (\underbrace{\Diamond \otimes \cdots \otimes \Diamond}_{r}) &
\end{array}
$$

$$
(A_1, \ldots, A_n, r) \to B \;\longmapsto\; \big(\phi(A_1), \ldots, \phi(A_n), \overbrace{\Diamond, \ldots, \Diamond}^{r}\big) \to \phi(B)
$$

[9] As already said: it is intuitively clear what 'corresponding' means here, i.e. the programs should be equal modulo all instructions dealing with $\Diamond$ only.

Note that ϕ is an injective map. We also allow ourselves to write $\phi(\Sigma)$ and $\phi(\Gamma)$ instead of $(\phi \circ \Sigma)$ and $(\phi \circ \Gamma)$ respectively.

To complete the construction of the LFPL-program we need to insert the necessary match and $\otimes$ instructions to decompose or compose these products of $\Diamond$'s. Finally operations consuming or recovering elements of $\Diamond$ like cons or $\mathsf{match}_{\mathsf{L}(\cdot)}$ need the proper arguments. Since the number of resources consumed or recovered by built-in functions is exactly the same in both languages, this process is straightforward to accomplish; so we do not give details here and merely state:

Lemma 2.3.2.

$$\forall \Sigma, \Gamma, n, e, A. \quad \Gamma, n \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\Sigma}} e : A \implies \phi(\Gamma), d_1 : \Diamond, \ldots, d_n : \Diamond \overset{\mathsf{LFPL}}{\vdash_{\phi(\Sigma)}} e' : \phi(A)$$

where the term e' is constructed according to term e as described above.

Transforming LFPL to $\widehat{\mathsf{LFPL}}$

By the way $\widehat{\mathsf{LFPL}}$ is constructed, a converse translation is only possible if the LFPL-program has a faithful signature, i.e. that the content of all result types of any occurring first-order types is zero. So throughout the following we consider programs with faithful signatures only. We leave unanswered whether the restriction to faithful signatures reduces expressive power. Although it seems likely to us that expressive power remains unaffected.

Definition 2.3.3. Define the map $\psi : \mathsf{LFPL}$ types $\longrightarrow \widehat{\mathsf{LFPL}}$ pure types as described below.

$$1 \longmapsto 1 \qquad \mathsf{N} \longmapsto \mathsf{N} \qquad \Diamond \longmapsto 1$$

$$\mathsf{L}(A) \longmapsto \mathsf{L}\big(\psi(A), \langle A \rangle\big) \qquad A \otimes B \longmapsto \psi(A) \otimes \psi(B)$$

$$\mathsf{T}(A) \longmapsto \mathsf{T}\big(\psi(A), \langle A \rangle\big) \qquad A + B \longmapsto \begin{cases} \big(\psi(A), \langle A \rangle - \langle B \rangle\big) + \psi(B) & \langle A \rangle \geq \langle B \rangle \\ \psi(A) + \big(\psi(B), \langle B \rangle - \langle A \rangle\big) & \text{otherwise} \end{cases}$$

$$(A_1, \ldots, A_n) \to B \longmapsto \big(\psi(A_1), \ldots, \psi(A_n), -\langle B \rangle + \textstyle\sum \langle A_i \rangle\big) \to \psi(B)$$

Again we extend the map point-wise to contexts Γ and signatures Σ:

$$\psi(\Gamma)(x) = (\psi \circ \Gamma)(x) = \psi(\Gamma(x)) \qquad \psi(\Sigma)(f) = (\psi \circ \Sigma)(x) = \psi(\Sigma(f))$$

Note that the presented map is not surjective, e.g. $\psi\big(\mathsf{L}(\Diamond \otimes \Diamond)\big) = \mathsf{L}(1 \otimes 1, 2) \neq \mathsf{L}(1, 2)$, since we simply replace each occurrence of $\Diamond$ by 1.

In order to complete the translation we also need a map of LFPL-terms to $\widehat{\mathsf{LFPL}}$-terms in accordance to the definition of ψ. We allow ourselves to reuse the symbol $|\cdot|$ to denote this map. Although $\widehat{\mathsf{LFPL}}$-terms are equally LF-terms, we shall not use this as a definition for $|\cdot| : \mathsf{LFPL} \to \mathsf{LF}$ by the arguments explained in Section 2.1, where we concluded that it is unacceptable that $|\cdot| : \mathsf{LFPL} \to \mathsf{LF}$ would map $\Diamond$ onto 1. However in the case of a translation from LFPL to $\widehat{\mathsf{LFPL}}$ there is no need to conceal the LFPL resource handling operations, as the $\widehat{\mathsf{LFPL}}$ translation contains this information anyway. Therefore simply replacing $\Diamond$ by 1 is suitable here.

Definition 2.3.4. Define $|\cdot|$: LFPL-terms $\longrightarrow$ $\widehat{\mathsf{LFPL}}$-terms inductively as follows:

Variable

$$|v| = \begin{cases} * & v : \Diamond \\ v & otherwise \end{cases}$$

Constant I + II

$$|*| = * \qquad |n| = n \qquad \text{(where } n \text{ is a integer constant)}$$

Standard Integer Infix Operator

$$|e_1 \star e_2| = |e_1| \star |e_2| \qquad (\text{where } \star \in \{+, -, =, \dots\})$$

Conditional

$$|\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3| = \mathsf{if}\ |e_1|\ \mathsf{then}\ |e_2|\ \mathsf{else}\ |e_3|$$

Pairing

$$|e_a \otimes e_b| = |e_a| \otimes |e_b|$$

Pair-Elimination

$$|\mathsf{match}\ e_1\ \mathsf{with}\ v_a \otimes v_b \Rightarrow e_2| = \mathsf{match}\ |e_1|\ \mathsf{with}\ v_a \otimes v_b \Rightarrow |e_2|$$

Inl, Inr

$$|\mathsf{inl}(e)| = \mathsf{inl}(|e|) \qquad |\mathsf{inr}(e)| = \mathsf{inr}(|e|)$$

Sum-Elimination

$$\begin{aligned} &|\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{inl}(v_a) \Rightarrow e_2\ |\mathsf{inr}(v_b) \Rightarrow e_3| \\ &\quad = \mathsf{match}\ |e_1|\ \mathsf{with}\ |\mathsf{inl}(v_a) \Rightarrow |e_2|\ |\mathsf{inr}(v_b) \Rightarrow |e_3| \end{aligned}$$

Empty List

$$|\mathsf{nil}| = \mathsf{nil}$$

List-Construction

$$|\mathsf{cons}(e_\Diamond, e_1, e_2)| = \mathsf{cons}(|e_1|, |e_2|)$$

List-Elimination

$$\begin{aligned}&|\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{nil} \Rightarrow e_2\ |\mathsf{cons}(e_\Diamond, e_h, e_t) \Rightarrow e_3| \\ &\quad = \mathsf{match}\ |e_1|\ \mathsf{with}\ |\mathsf{nil} \Rightarrow |e_2|\ |\mathsf{cons}(e_h, e_t) \Rightarrow |e_3|\end{aligned}$$

Leaf

$$|\mathsf{leaf}| = \mathsf{leaf}$$

Tree-Node

$$|\mathsf{node}(e_\Diamond, e_a, e_l, e_r,)| = \mathsf{node}(|e_a|, |e_l|, |e_r|)$$

Tree-Elimination

$$\begin{aligned}&|\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{leaf} \Rightarrow e_2\ |\mathsf{node}(v_\Diamond, v_a, v_l, v_r) \Rightarrow e_3| \\ &\quad = \mathsf{match}\ |e_1|\ \mathsf{with}\ |\mathsf{leaf} \Rightarrow |e_2|\ |\mathsf{node}(v_a, v_l, v_r) \Rightarrow |e_3|\end{aligned}$$

Function Application

$$|f_i\,(e_1, \ldots, e_n)| = f_i\big(|e_1|, \ldots, |e_n|\big)$$

Now we are ready to prove that these maps give rise to a sensible translation from LFPL to $\widehat{\mathsf{LFPL}}$:

Lemma 2.3.5. *Assume that Σ is a faithful* LFPL*-signature, then*

$$\forall \Gamma, e, A\,.\ \Gamma \overset{\mathsf{LFPL}}{\vdash_\Sigma} e : A \implies \psi(\Gamma), \langle\Gamma\rangle - \langle A\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e| : \psi(A)$$

Proof. We prove the claim by induction on the composition of LFPL terms, but note first that in each case we already know that $\langle\Gamma\rangle - \langle A\rangle \geq 0$ by Lemma 2.1.3, since we assumed Σ to be faithful. Hence we concentrate on the validity of the constraints of the $\widehat{\mathsf{LFPL}}$ type rules and do not mention Lemma 2.1.3 explicitly anymore when making use of the induction hypothesis.

Variable: $\Gamma \overset{\mathsf{LFPL}}{\vdash_\Sigma} v : A$

We have to distinguish two cases, but in both cases the required $\widehat{\mathsf{LFPL}}$ type rule has no constraints.

$A \neq \Diamond$ By $\Gamma(v) = A$ we deduce $\psi(\Gamma)(v) = \psi(A)$ and by definition $\langle\Gamma\rangle \geq \langle\Gamma(v)\rangle = \langle A\rangle$. So the claim is obviously true as $|v| = v$.

$A = \Diamond$ In this case $|v| = *$ and we may deduce

$$\psi(\Gamma), \langle\Gamma\rangle - 1 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} * : \mathsf{1}$$

since again by Lemma 2.1.3 we know that $\langle\Gamma\rangle - 1 \geq 0$ as already mentioned above.

Constant: $\Gamma \overset{\mathsf{LFPL}}{\vdash_\Sigma} n : \mathsf{N}$

Trivially we derive

$$\psi(\Gamma), \langle\Gamma\rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} n : \mathsf{N}$$

as required.

Integer Infix: $\Gamma_1, \Gamma_2 \overset{\mathsf{LFPL}}{\vdash_\Sigma} e_1 \star e_2 : \mathsf{N}$

By the induction hypothesis

$$\psi(\Gamma_1), \langle\Gamma_1\rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_1| : \mathsf{N}$$
$$\psi(\Gamma_2), \langle\Gamma_2\rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_2| : \mathsf{N}$$

thus we easily conclude

$$\psi(\Gamma_1, \Gamma_2), \langle\Gamma_1, \Gamma_2\rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_1| \star |e_2| : \mathsf{N}$$

Conditional: $\Gamma_1, \Gamma_2 \overset{\mathsf{LFPL}}{\vdash_\Sigma} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A$

By the LFPL-typing rules and the induction hypothesis follows

$$\psi(\Gamma_1), \langle\Gamma_1\rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_1| : \mathsf{N}$$
$$\psi(\Gamma_2), \langle\Gamma_2\rangle - \langle A\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_2| : \psi(A)$$
$$\psi(\Gamma_2), \langle\Gamma_2\rangle - \langle A\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_3| : \psi(A)$$

hence we derive

$$\psi(\Gamma_1, \Gamma_2), \langle \Gamma_1, \Gamma_2 \rangle - \langle A \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} \mathsf{if}\ |e_1|\ \mathsf{then}\ |e_2|\ \mathsf{else}\ |e_3| : \psi(A)$$

as clearly

$$\langle \Gamma_1, \Gamma_2 \rangle - \langle A \rangle \geq \big(\langle \Gamma_1 \rangle - 0\big) + \big(\langle \Gamma_2 \rangle - \langle A \rangle\big)$$

holds.

Pairing: $\Gamma_1, \Gamma_2 \overset{\mathsf{LFPL}}{\vdash_\Sigma} e_1 \otimes e_2 : A_1 \otimes A_2$

Again, we call upon the induction hypothesis to obtain

$$\psi(\Gamma_1), \langle \Gamma_1 \rangle - \langle A_1 \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_1| : \psi(A_1)$$
$$\psi(\Gamma_2), \langle \Gamma_2 \rangle - \langle A_2 \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_2| : \psi(A_2)$$

and conclude

$$\psi(\Gamma_1, \Gamma_2), \langle \Gamma_1, \Gamma_2 \rangle - \langle A_1 \otimes A_2 \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_1| \otimes |e_2| : \psi(A_1) \otimes \psi(A_2)$$

since $\langle A_1 \otimes A_2 \rangle = \langle A_1 \rangle + \langle A_2 \rangle$ by definition.

Pair-Elimination: $\Gamma_1, \Gamma_2 \overset{\mathsf{LFPL}}{\vdash_\Sigma} \mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2 : C$

We assume that $\Gamma_1 \overset{\mathsf{LFPL}}{\vdash_\Sigma} e_1 : A_1 \otimes A_2$. By appeal to the typing rules and the induction hypothesis we deduce

$$\psi(\Gamma_1), \langle \Gamma_1 \rangle - \langle A_1 \otimes A_2 \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_1| : \psi(A_2) \otimes \psi(A_1)$$
$$\psi(\Gamma_2, v_1 : A_1, v_2 : A_2), \langle \Gamma_2, v_1 : A_1, v_2 : A_2 \rangle - \langle C \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_2| : \psi(C)$$

and observe that

$$\langle \Gamma_1, \Gamma_2 \rangle - \langle C \rangle \geq \big(\langle \Gamma_1 \rangle - \langle A_1 \otimes A_2 \rangle\big) + \big(\langle \Gamma_2, v_1 : A_1, v_2 : A_2 \rangle - \langle C \rangle\big)$$

holds by the point-wise extension of Definition 2.1.1 to typing contexts and therefore we conclude as needed

$$\psi(\Gamma_1, \Gamma_2), \langle \Gamma_1, \Gamma_2 \rangle - \langle C \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} \mathsf{match}\ |e_1|\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow |e_2| : \psi(C)$$

Inl, Inr: $\Gamma \vdash_\Sigma \mathsf{inl}(e) : A + B$

Using the induction hypothesis we derive

$$\psi(\Gamma), \langle\Gamma\rangle - \langle A\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash}_{\psi(\Sigma)} |e| : \psi(A)$$

If $\langle A\rangle \geq \langle B\rangle$ then $\psi(A+B) = \big(\psi(A), \langle A\rangle - \langle B\rangle\big) + \psi(B)$ and we derive

$$\psi(\Gamma), \langle\Gamma\rangle - \langle B\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash}_{\psi(\Sigma)} \mathsf{inl}(|e|) : \big(\psi(A), \langle A\rangle - \langle B\rangle\big) + \psi(B)$$

by the $\widehat{\mathsf{LFPL}}$ type rule **Inl II** since clearly $\langle\Gamma\rangle - \langle B\rangle \geq \big(\langle\Gamma\rangle - \langle A\rangle\big) + \big(\langle A\rangle - \langle B\rangle\big)$. Otherwise, if $\langle A\rangle < \langle B\rangle$ we straightforwardly apply the rule **Inl I**.

$$\psi(\Gamma), \langle\Gamma\rangle - \langle A\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash}_{\psi(\Sigma)} \mathsf{inl}(|e|) : \psi(A) + \big(\psi(B), \langle A\rangle - \langle B\rangle\big)$$

The case of $\Gamma \overset{\mathsf{LFPL}}{\vdash}_\Sigma \mathsf{inr}(e) : A + B$ is analogue.

Sum-Elimination: $\Gamma_1, \Gamma_2 \overset{\mathsf{LFPL}}{\vdash}_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ \mid\mathsf{inl}(v) \Rightarrow e_2\ \mid\mathsf{inr}(v) \Rightarrow e_3 : C$

Let $\Gamma_1 \overset{\mathsf{LFPL}}{\vdash}_\Sigma e_1 : A + B$. We distinguish:

Case $\langle A\rangle \geq \langle B\rangle$: By the typing rules and the induction hypothesis we have

$$\begin{aligned}
\psi(\Gamma_1), \langle\Gamma_1\rangle - \langle B\rangle &\overset{\widehat{\mathsf{LFPL}}}{\vdash}_{\psi(\Sigma)} |e_1| : \big(\psi(A), \langle A\rangle - \langle B\rangle\big) + \psi(B) \\
\psi(\Gamma_2, v : A), \langle\Gamma_2, v : A\rangle - \langle C\rangle &\overset{\widehat{\mathsf{LFPL}}}{\vdash}_{\psi(\Sigma)} |e_2| : \psi(C) \\
\psi(\Gamma_2, v : B), \langle\Gamma_2, v : B\rangle - \langle C\rangle &\overset{\widehat{\mathsf{LFPL}}}{\vdash}_{\psi(\Sigma)} |e_3| : \psi(C)
\end{aligned}$$

In order to apply the type rule **Sum-Elimination II**, we note that the inequalities

$$\begin{aligned}
\langle\Gamma_1, \Gamma_2\rangle - \langle C\rangle &\geq \big(\langle\Gamma_1\rangle - \langle B\rangle\big) + \big(\langle\Gamma_2, v : A\rangle - \langle C\rangle\big) - \big(\langle A\rangle - \langle B\rangle\big) \\
\langle\Gamma_1, \Gamma_2\rangle - \langle C\rangle &\geq \big(\langle\Gamma_1\rangle - \langle B\rangle\big) + \big(\langle\Gamma_2, v : B\rangle - \langle C\rangle\big)
\end{aligned}$$

are valid, and deduce as required

$$\begin{aligned}
\psi(\Gamma_1, \Gamma_2), \langle\Gamma_1, \Gamma_2\rangle - \langle C\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash}_{\psi(\Sigma)} \mathsf{match}\ |e_1|\ \mathsf{with}\ &: \psi(C) \\
\mid\mathsf{inl}\,(v) &\Rightarrow |e_2| \\
\mid\mathsf{inr}(v) &\Rightarrow |e_3|
\end{aligned}$$

Case $\langle A \rangle < \langle B \rangle$**:** is a symmetrical analogue.

Nil: $\Gamma \overset{\mathsf{LFPL}}{\vdash_\Sigma} \mathsf{nil} : \mathsf{L}(A)$

Trivial, as the $\widehat{\mathsf{LFPL}}$ type judgement

$$\psi(\Gamma), \langle \Gamma \rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} \mathsf{nil} : \mathsf{L}(\psi(A), \langle A \rangle)$$

holds without any constraints.

List-Construction: $\Gamma_d, \Gamma_h, \Gamma_t \overset{\mathsf{LFPL}}{\vdash_\Sigma} \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)$

By the LFPL typing rules and by use of the induction hypothesis we obtain

$$\begin{aligned} \psi(\Gamma_d), \langle \Gamma_d \rangle - 1 &\overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_d| : 1 \\ \psi(\Gamma_h), \langle \Gamma_h \rangle - \langle A \rangle &\overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_h| : \psi(A) \\ \psi(\Gamma_t), \langle \Gamma_t \rangle - 0 &\overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_t| : \mathsf{L}(\psi(A), \langle A \rangle) \end{aligned}$$

Hence we derive

$$\langle \Gamma_d, \Gamma_h, \Gamma_t \rangle \geq \langle \Gamma_h \rangle + \langle \Gamma_t \rangle + (1 + \langle A \rangle)$$

and therefore conclude

$$\psi(\Gamma_d, \Gamma_h, \Gamma_t), \langle \Gamma_d, \Gamma_h, \Gamma_t \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} \mathsf{cons}(|e_h|, |e_t|) : \mathsf{L}(\psi(A), \langle A \rangle)$$

as $\langle \mathsf{L}(\psi(A), \langle A \rangle) \rangle = 0$.

List-Elimination: $\Gamma_1, \Gamma_2 \overset{\mathsf{LFPL}}{\vdash_\Sigma} \mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{nil} \to e_2\ |\mathsf{cons}(d, h, t) \to e_3 : C$

By the LFPL typing rules – assuming that $\Gamma_1 \overset{\mathsf{LFPL}}{\vdash_\Sigma} e_1 : \mathsf{L}(A)$ – and the induction hypothesis we have

$$\begin{aligned} \psi(\Gamma_1), \langle \Gamma_1 \rangle - 0 &\overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_1| : \mathsf{L}(\psi(A), \langle A \rangle) \\ \psi(\Gamma_2), \langle \Gamma_2 \rangle - \langle C \rangle &\overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_2| : \psi(C) \\ \psi(\Delta), \langle \Delta \rangle - \langle C \rangle &\overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_3| : \psi(C) \end{aligned}$$

where $\Delta = \Gamma_2, d : \Diamond, h : A, t : \mathsf{L}(A)$. We observe that by definition

$$\langle \Delta \rangle = \langle \Gamma_2 \rangle + \langle \Diamond \rangle + \langle A \rangle + \langle \mathsf{L}(A) \rangle = \langle \Gamma_2 \rangle + \big(1 + \langle A \rangle\big)$$

So we derive

$$\langle \Gamma_1, \Gamma_2 \rangle - \langle C \rangle \geq \langle \Gamma_1 \rangle + \big(\langle \Gamma_2 \rangle - \langle C \rangle\big)$$
$$\langle \Gamma_1, \Gamma_2 \rangle - \langle C \rangle \geq \langle \Gamma_1 \rangle + \big(\langle \Delta \rangle - \langle C \rangle\big) - \big(1 + \langle A \rangle\big)$$

which allows the desired type judgement

$$\psi(\Gamma_1, \Gamma_2), \langle \Gamma_1, \Gamma_2 \rangle - \langle C \rangle \quad \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} \quad \begin{array}{l} \mathsf{match}\ |e_1|\ \mathsf{with} \\ \quad \mid \mathsf{nil} \Rightarrow |e_2| \\ \quad \mid \mathsf{cons}(h, t) \Rightarrow |e_3| \end{array} : \psi(C)$$

Leaf: $\Gamma \overset{\mathsf{LFPL}}{\vdash_\Sigma} \mathsf{leaf} : \mathsf{T}(A)$

By the $\widehat{\mathsf{LFPL}}$ typing rules we derive directly

$$\psi(\Gamma), \langle \Gamma \rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} \mathsf{leaf} : \mathsf{T}(\psi(A), \langle A \rangle)$$

Node: $\Gamma_d, \Gamma_a, \Gamma_l, \Gamma_r \overset{\mathsf{LFPL}}{\vdash_\Sigma} \mathsf{node}(e_d, e_a, e_l, e_r) : \mathsf{T}(A)$

The LFPL typing rules and the application of the induction hypothesis yields

$$\psi(\Gamma_d), \langle \Gamma_d \rangle - 1 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_d| : 1$$
$$\psi(\Gamma_a), \langle \Gamma_a \rangle - \langle A \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_a| : \psi(A)$$
$$\psi(\Gamma_l), \langle \Gamma_l \rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_l| : \mathsf{T}(\psi(A), \langle A \rangle)$$
$$\psi(\Gamma_r), \langle \Gamma_r \rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_r| : \mathsf{T}(\psi(A), \langle A \rangle)$$

therefore the inequality

$$\langle \Gamma_d, \Gamma_a, \Gamma_l, \Gamma_r \rangle \geq \big(1 + \langle A \rangle\big) + \langle \Gamma_a \rangle + \langle \Gamma_l \rangle + \langle \Gamma_r \rangle$$

holds, allowing us to deduce

$$\psi(\Gamma_d, \Gamma_a, \Gamma_l, \Gamma_r), \langle \Gamma_d, \Gamma_a, \Gamma_l, \Gamma_r \rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} \mathsf{node}(|e_a|, |e_l|, |e_r|) : \mathsf{T}(\psi(A), \langle A \rangle)$$

Tree-Elimination: $\Gamma_1, \Gamma_2 \overset{\mathsf{LFPL}}{\vdash_\Sigma} \mathsf{match}\ e_1\ \mathsf{with} \mid \mathsf{leaf} \Rightarrow e_2 \mid \mathsf{node}(d, a, l, r) \Rightarrow e_3 : C$

By the LFPL typing rules and the induction hypothesis we obtain

$$\psi(\Gamma_1), \langle \Gamma_1 \rangle - 0 \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_1| : \mathsf{T}(\psi(A), \langle A \rangle)$$
$$\psi(\Gamma_2), \langle \Gamma_2 \rangle - \langle C \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_2| : \psi(C)$$
$$\psi(\Delta), \langle \Delta \rangle - \langle C \rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_3| : \psi(C)$$

where $\Delta = \Gamma_2, d : \Diamond, a : A, l : \mathsf{T}(A), r : \mathsf{T}(A)$ and observe that $\langle\Delta\rangle = \langle\Gamma_2\rangle + \big(1 + \langle A\rangle\big)$ by the definition of $\langle\cdot\rangle$. Hence

$$\langle\Gamma_1, \Gamma_2\rangle - \langle C\rangle \geq \langle\Gamma_1\rangle + \langle\Gamma_2\rangle - \langle C\rangle$$
$$\langle\Gamma_1, \Gamma_2\rangle - \langle C\rangle \geq \langle\Gamma_1\rangle + \langle\Delta\rangle - \langle C\rangle - \big(1 + \langle A\rangle\big)$$

and thus we may derive

$$\psi(\Gamma_1, \Gamma_2), \langle\Gamma_1, \Gamma_2\rangle - \langle C\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} \mathsf{match}\ |e_1|\ \mathsf{with}\ \begin{array}{ll} \mid \mathsf{leaf} \Rightarrow |e_2| & : \psi(C) \\ \mid \mathsf{node}(a, l, r) \Rightarrow |e_3| & \end{array}$$

Function Application: $\Gamma_1, \ldots, \Gamma_n \overset{\mathsf{LFPL}}{\vdash_\Sigma} f(e_1, \ldots, e_n) : B$

Assume that $\Sigma(f) = (A_1, \ldots, A_n) \to B$. By the induction hypothesis we obtain for $i = 1, \ldots, n$ that

$$\psi(\Gamma_i), \langle\Gamma_i\rangle - \langle A_i\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} |e_i| : \psi(A_i)$$

Obviously the equation

$$\langle\Gamma_1, \ldots, \Gamma_n\rangle - \langle B\rangle = \Big(-\langle B\rangle + \sum \langle A_i\rangle\Big) + \sum\Big(\langle\Gamma_i\rangle - \langle A_i\rangle\Big)$$

holds and we can finally conclude

$$\psi(\Gamma_1, \ldots, \Gamma_n), \langle\Gamma_1, \ldots, \Gamma_n\rangle - \langle B\rangle \overset{\widehat{\mathsf{LFPL}}}{\vdash_{\psi(\Sigma)}} f(|e_1|, \ldots, |e_n|) : \psi(B)$$

since by definition of ψ we know that

$$\psi(\Sigma)(f) = (\psi(A_1), \ldots, \psi(A_n), -\langle B\rangle + \sum \langle A_i\rangle) \to \psi(B)$$

One may note that $\langle B\rangle = 0$, i.e. that Σ is a faithful signature, is not even necessary in this particular step, though it is of course required for the application of Lemma 2.1.3, hidden within each use of the induction hypothesis. □

So it remains to be shown that the obtained translation indeed 'corresponds' to the program we started with. This can only be proved by the use of some accepted formal semantics for both languages. In this case the semantics should take into account the resource usage of the programs, hence the set-theoretic semantics of LFPL and $\widehat{\mathsf{LFPL}}$ given in Definition A.2.2 and Definition 2.2.3 cannot be used for this purpose.

Regrettably, designing some suitable semantics with this intention in mind is beyond the scope of this work. Thus we are content by giving the translations of both

languages into LF, which is equivalent to use of the given set-theoretic semantics. Of course, these translations must be trusted to preserve the meaning of a program again. We therefore do not complete the proof of equivalence of $\widehat{\mathsf{LFPL}}$ and the faithful fragment of $\widehat{\mathsf{LFPL}}$ here, but merely conjecture it in presence of the Lemmata 2.3.2 and 2.3.5 and by the simple construction of the translations.

We also avoid defining the map $|\cdot| : \mathsf{LFPL} \longrightarrow \mathsf{LF}$, but let us briefly discuss at this point the problems faced by constructing it in accordance to what was said in section 2.1. The map $|\cdot| : \mathsf{LFPL} \longrightarrow \mathsf{LF}$ must respect the 'normalising' of the resource handling of LFPL programs. Thus replacing the LFPL type A by its 'normalised' form[10] $\phi \circ \psi(A) \otimes (\underbrace{\Diamond \otimes \cdots \otimes \Diamond}_{\langle A \rangle})$ within a program must not alter its translation to LF. So $|\cdot| : \mathsf{LFPL} \to \mathsf{LF}$ must eliminate operations transforming a type into this 'normalised' form, while retaining all other operations. How to distinguish those cases is not clear, e.g. consider a program P that contains the type $\big((\Diamond \otimes 1) + (\mathsf{N} \otimes \Diamond)\big) \otimes \mathsf{N}$ and turn it into a program $\tilde{P}$ that only uses the 'normal' form of this type, namely $\big((1 + \mathsf{N}) \otimes \mathsf{N}\big) \otimes \Diamond$. This can be easily done by inserting the proper match, inl, inr and $\otimes$ operations, and since we require $|P| = |\tilde{P}|$, the $|\cdot|$-map must eliminate such type conversions. But those type conversions are not easy to distinguish from ordinary instructions, as for example the first Pair-Elimination has nothing to do with the resource type at first glance. In addition, $|\cdot|$ must not leave artifacts of the like $\mathsf{match}\ e\ \mathsf{with}\ u \otimes v \Rightarrow u \otimes v$ when they were not present in the original program, as this would prevent us from comparing programs by α-equivalence only.

On the other hand defining the map $|\cdot| : \widehat{\mathsf{LFPL}} \longrightarrow \mathsf{LF}$, which we require anyway, is straightforward to accomplish. The problem of type conversions, with respect to the resource type only, does not arise in this case naturally, since $\widehat{\mathsf{LFPL}}$ already ensures a uniform handling of the resources.

Definition 2.3.6. For any $\widehat{\mathsf{LFPL}}$-term e let $|e| = e$, as there are no differences in the term grammar of LF and $\widehat{\mathsf{LFPL}}$.

For the $\widehat{\mathsf{LFPL}}$-types we define:

$$|1| = 1 \qquad |(A, n)| = A \qquad |\mathsf{N}| = \mathsf{N}$$

$$|A \otimes B| = |A| \otimes |B| \qquad |A + B| = |A| + |B|$$

$$|\mathsf{L}(A)| = \mathsf{L}(|A|) \qquad |\mathsf{T}(A)| = \mathsf{T}(|A|)$$

[10] for a true normal form ψ and ϕ must be altered to eliminate unnecessary occurrences of type 1

$$|f(A_1, \ldots, A_m, n) \to B| = f(|A_1|, \ldots, |A_m|) \to |B|$$

Lemma 2.3.7.

$$\Gamma, n \overset{\widehat{\mathsf{LFPL}}}{\vdash_\Sigma} e : C \implies |\Gamma| \overset{\mathsf{LF}}{\vdash_{|\Sigma|}} e : |C|$$

Proof. Trivial, as each LF typing rule is a weakened form of its corresponding $\widehat{\mathsf{LFPL}}$ typing rule. □

2.4 The transformation from LF to $\widehat{\mathsf{LFPL}}$

We will now examine the process transforming a LF- into a $\widehat{\mathsf{LFPL}}$-program. Since the term grammars of both languages are identical, the problem solely reduces to derive a $\widehat{\mathsf{LFPL}}$-signature $\hat{\Sigma}$ so that the program is well-typed. Unfortunately, the following theorem shows that this is not necessarily an easy task:

The complexity of the $\widehat{\mathsf{LFPL}}$-Signature Problem

Theorem 2.4.1. *Let P be a* LF*-program with signature Σ. Finding a $\widehat{\mathsf{LFPL}}$-signature $\hat{\Sigma}$ with $\left|\hat{\Sigma}\right| = \Sigma$ so that P is a well-typed $\widehat{\mathsf{LFPL}}$-program is an $\mathcal{NP}$-hard task.*

We will prove the theorem by reducing the problem 3SAT to our task, for which is well-known that it is $\mathcal{NP}$-complete. We will pave the way for the proof first:

Definition 2.4.2. Let $\Phi = (u_{11} \vee u_{12} \vee u_{13}) \wedge \cdots \wedge (u_{n1} \vee u_{n2} \vee u_{n3})$ be an instance of 3SAT, where u_{kj} equals either v_i or $\neg v_i$. We define the LF-program P_Φ and its according signature Σ_Φ in the following way:

- For each occurring variable $x_i \in \Phi$ define

$$f_i(*) := f_i(*)$$
$$\Sigma_\Phi(f_i) := \mathsf{1} \to \mathsf{1} + \mathsf{1}$$
$$h_i(v) := h_i \left(h_i \left(\begin{array}{l} \mathsf{match}\ f_i(*)\ \mathsf{with} \\ \quad \mid \mathsf{inl}\,(s_1) \Rightarrow \mathsf{nil} \\ \quad \mid \mathsf{inr}(s_1) \Rightarrow \mathsf{match}\ f_i(*)\ \mathsf{with} \\ \qquad \mid \mathsf{inl}\,(s_2) \Rightarrow \mathsf{cons}(*, \mathsf{nil}) \\ \qquad \mid \mathsf{inr}(s_2) \Rightarrow \mathsf{nil} \end{array} \right) \right)$$
$$\Sigma_\Phi(h_i) := \mathsf{L}(\mathsf{1}) \to \mathsf{L}(\mathsf{1})$$

- For each of the n clauses in Φ define

$$g_i(v) := g_i\left(g_i\left(cu_{i1}\left[w \setminus cu_{i2}\left[w \setminus cu_{i3}\left[w \setminus \mathsf{cons}(*,\mathsf{nil})\right]\right]\right]\right)\right)$$

$$\Sigma_\Phi(g_i) := \mathsf{L}(1) \to \mathsf{L}(1)$$

where

$$cu_{ij} := \begin{cases} \mathsf{match}\ f_k(*)\ \mathsf{with}\ \mid \mathsf{inl}(s_j) \to \mathsf{nil}\ \mid \mathsf{inr}(s_j) \to w & \text{if } u_{ij} = v_k \\ \mathsf{match}\ f_k(*)\ \mathsf{with}\ \mid \mathsf{inl}(s_j) \to w\ \mid \mathsf{inr}(s_j) \to \mathsf{nil} & \text{if } u_{ij} = \neg v_k \end{cases}$$

Example 2.4.3. Since we can consider each clause of Φ separately to construct P_Φ, we consider as an example the single clause

$$v_1 \vee \neg v_2 \vee v_3$$

and show how to construct g_1. Adding further clauses will similarly result in adding more functions $g_2, g_3, \ldots$

$$g_1(v) := g_1\left(g_1\left(\begin{array}{l} \mathsf{match}\ f_1(*)\ \mathsf{with} \\ \quad \mid \mathsf{inl}(s_1) \Rightarrow \mathsf{nil} \\ \quad \mid \mathsf{inr}(s_1) \Rightarrow \mathsf{match}\ f_2(*)\ \mathsf{with} \\ \qquad \mid \mathsf{inl}(s_2) \Rightarrow \mathsf{match}\ f_3(*)\ \mathsf{with} \\ \qquad\quad \mid \mathsf{inl}(s_3) \Rightarrow \mathsf{nil} \\ \qquad\quad \mid \mathsf{inr}(s_3) \Rightarrow \mathsf{cons}(*, \mathsf{nil}) \\ \qquad \mid \mathsf{inr}(s_2) \Rightarrow \mathsf{nil} \end{array}\right)\right)$$

Note that for each x_j the function h_j is equal to the function g corresponding to the clause $v_j \vee \neg v_j$, but we will examine this and the complete intention of P_Φ closely in the proof of Lemma 2.4.5.

Lemma 2.4.4. *The program P_Φ can be constructed within polynomial time depending on the length of Φ.*

Proof. Let l denote the length of Φ.

There are at most l different variables, requiring us to define l functions f_i and h_i respectively. Every definition can be done within constant time c_1.

There are at most l different clauses, each containing at most l literals. Let c_2 denote the time needed to write down one cu_{ij} and c_3 the time needed for defining the header

and the trailing cons-operation in each function definition. The substitutions do not raise the computing time, as w occurs in each cu_{ij} exactly once. The substitutions merely provide a convenient notation, but they are not a part of the construction.

This sums up to a computing time of at most $c_1x + c_2x^2 + c_3x$ steps for P_Φ. □

Lemma 2.4.5. *A 3SAT formula Φ is satisfiable if and only if there is a $\widehat{\mathsf{LFPL}}$-signature $\hat{\Sigma}$ with $\left|\hat{\Sigma}\right| = \Sigma_\Phi$ so that P_Φ is a well-typed $\widehat{\mathsf{LFPL}}$-program in the presence of $\hat{\Sigma}$.*

Proof. Let $\hat{\Sigma}$ be such a $\widehat{\mathsf{LFPL}}$-signature. By the way f_i and h_i are constructed, either

$$\hat{\Sigma}(f_i) = (\mathsf{1}, 0) \to 1 + (1, n) \qquad \text{or} \qquad \hat{\Sigma}(f_i) = (\mathsf{1}, 0) \to (1, n) + 1$$

for some $n > 0$. The first possibility representing v_i being true and the other v_i being false. That either one of these two possibilities is due is enforced by the definition of h_i: There is always one branch that consumes at least one resource (the one containing the cons-operation), which must come from the Sum-Elimination operations as any resources obtained from the functions arguments are consumed by the two recursive calls of h_i to itself. It is not clear which of the two Sum-Elimination operations delivers the resource needed. So each h_i represents the tautologic clause $v_i \vee \neg v_i$.

Thus define

$$\rho(v_i) := \begin{cases} true & \hat{\Sigma}(f_i) = (\mathsf{1}, 0) \to \mathsf{1} + (\mathsf{1}, n) \quad n \in \mathbb{N}^+ \\ false & \hat{\Sigma}(f_i) = (\mathsf{1}, 0) \to (\mathsf{1}, n) + \mathsf{1} \quad n \in \mathbb{N}^+ \end{cases}$$

As each g_i is well-typed under $\hat{\Sigma}$, the path leading to the trailing cons-operation must free at least one resource from one of the preceding Sum-Elimination operations, since again all resources possibly obtained from the arguments of g_i are in turn consumed by the recursive calls (In fact, g_i cannot obtain any resources from its arguments, since it would require the double amount of resources to satisfy the recursive calls).

Since the path leading to the cons-operation was constructed in correspondence to the signs of the literals, at least one literal in clause i must therefore evaluate to true under ρ.

The logical 'and' connection between all clauses of Φ results from the fact that each defining clause for the g_i must be well-defined under the same signature $\hat{\Sigma}$. So Φ_ρ must evaluate to true when $\hat{\Sigma}$ types P_Φ.

On the converse, if there is a valuation ρ such that Φ_ρ evaluates to true, then define

$$\hat{\Sigma}(f_i) := \begin{cases} (1,0) \to \mathsf{1} + (1,1) & \rho(v_i) = true \\ (1,0) \to (1,1) + \mathsf{1} & \rho(v_i) = false \end{cases}$$
$$\hat{\Sigma}(f_i) := (\mathsf{L}(1,0), 0) \to \mathsf{L}(1,0)$$
$$\hat{\Sigma}(f_i) := (\mathsf{L}(1,0), 0) \to \mathsf{L}(1,0)$$

under which P_Φ is well-typed, as in each g_i at least one resource is freed along the path to the cons-operation, because at least one literal is true in clause i. □

Proof of Theorem 2.4.1. Using Definition 2.4.2 we can transform every instance of 3SAT in an equivalent $\widehat{\mathsf{LFPL}}$ typing problem by Lemma 2.4.5. This transformation can be completed within polynomial time by Lemma 2.4.4. Hence 3SAT reduces to solving $\widehat{\mathsf{LFPL}}$ typing problems. □

It is true that from a semantic viewpoint P_Φ is a bit of a nuisance as its execution cannot terminate in any way. The functions f_i have the same purpose as the function `borr` presented in Section 2.1. We explicitly tried to exclude such functions within $\widehat{\mathsf{LFPL}}$, but it seems hard to exclude such programs by syntactical means alone without restricting expressive power to primitive recursion. Further work has to be done here and it might turn out that the sum-type problem is feasible when it is possible to restrict the language to sensible function definitions that cannot try to borrow resources from non-termination.

Since it is our goal to derive a feasible integer linear program for a given LF-program P, named again $(\!|P|\!)$ for reference, that is solvable if and only if there is an equivalent $\widehat{\mathsf{LFPL}}$-program $\hat{P}$, we will eliminate the problem of deciding the order of sums by simply ignoring it. We will always decide to have the rich type on the right side of a sum. In this way we might miss a sensible solution, but we can ensure feasibility. In the case that we do not derive a solution at all we are forced to swap some of the occurring sum-types in the LF-program P and try again. Trying out all possible combinations leads in the worst case to exponential run-time, exponential in the number of sum-types occurring. Since the number of occurring sum-types with a necessarily rich component other than (A,0) can be reasonably expected to be small compared to a program's size, we do not consider this as a sever disadvantage.

Another imaginable disadvantage of this ignorant approach to the problem might be obtaining a solution that is not optimal though still a solution, e.g. obtaining a rich subtype $\big(B + (A,0), n\big)$ where $\big(A + (B,m), n-m\big)$ would be sufficient but not

necessary. But it will turn out later on that the notion of optimality is controversial anyway.

Definition 2.4.6. Let $\widehat{\mathsf{LFPL}}_R$ denote the fragment of $\widehat{\mathsf{LFPL}}$ that does not contain the type $R + P$, hence we also reject the typing rules (Inl II), (Inr II) and (Sum-Elimination II).

Building the ILP for $\widehat{\mathsf{LFPL}}_R$

In order to derive the $\widehat{\mathsf{LFPL}}_R$ signature for a given LF program P, we will consider all possible $\widehat{\mathsf{LFPL}}_R$ type derivations for P. Since the defining terms of P remain unchanged by a translation to $\widehat{\mathsf{LFPL}}_R$, the structure of the type derivation is already fixed as we will see. Thus we leave all occurring resource parameters unknown and gather the arising constraints of the skeleton of the $\widehat{\mathsf{LFPL}}_R$ type derivation. Whenever a new value would be needed within the premises of a type rule to be applied, we simply introduce a fresh variable ranging over $\mathbb{N}$. Each instance solving all the constraints then gives rise to a valid $\widehat{\mathsf{LFPL}}_R$ type derivation for P. The overall principle of this technique appeared probably first in a work by J. Palsberg and M. I. Schwartzbach [P&S91], though they faced more complex constraints than just integer linear inequalities as we will do.

We start by constructing a general $\widehat{\mathsf{LFPL}}_R$ type-scheme for each LF type, whose instantiations shall cover all possible $\widehat{\mathsf{LFPL}}_R$ type translations that a particular LF type may have. By an $\widehat{\mathsf{LFPL}}_R$ *type-scheme* we simply mean any $\widehat{\mathsf{LFPL}}_R$ type were some of resource parameters are unknown and replaced by variables ranging over $\mathbb{N}$, e.g. (N, v), $\mathsf{L}(\mathsf{N}, v)$, $\mathsf{1} + (\mathsf{1}, v)$, $(\mathsf{L}(\mathsf{N}, 2), \mathsf{N}, v) \multimap \mathsf{L}(\mathsf{1}, w)$ are all $\widehat{\mathsf{LFPL}}_R$ type-schemes when v, w are variables ranging over the natural numbers. Note that any $\widehat{\mathsf{LFPL}}_R$ type is an $\widehat{\mathsf{LFPL}}_R$ type-scheme as well.

Definition 2.4.7. $[\cdot] : \mathsf{LF}\text{-types} \to \widehat{\mathsf{LFPL}}\text{-type schemes}$:

$$
\begin{aligned}
[\mathsf{1}] &= \mathsf{1} \\
[\mathsf{N}] &= \mathsf{N} \\
[A \otimes B] &= [A] \otimes [B] \\
[A + B] &= [A] + ([B], x) \\
[\mathsf{L}(A)] &= \mathsf{L}([A], x) \\
[\mathsf{T}(A)] &= \mathsf{T}([A], x) \\
[(A_1, \ldots, A_n) \multimap B] &= \big([A_1], \ldots, [A_n], x\big) \multimap [B]
\end{aligned}
$$

where x is a fresh positive integer variable. We say that $[A]$ is the *enriched type* or type-scheme of A. The map is clearly injective.

Extend the map further to LF-signatures by $[\Sigma]\,(f) := [\Sigma(f)]$. Of course we have to be more careful by choosing the names for the fresh integer variables: As we want to refer multiple times to a signature, the names of the fresh integer variables chosen must always be the same in subsequent consultations of the same enriched signature. Additionally no variable name must be shared between the enriched type of two different functions belonging to the same signature.

Example 2.4.8. As an example we provide the enriched signature of the insertion sort algorithm, whose LF implementation was presented in Example 1.8:

$$\Sigma_{IS}(\mathtt{sort}) = (\mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N}) \qquad [\Sigma_{IS}]\,(\mathtt{sort}) = (\mathsf{L}(\mathsf{N}, l_1)\,, x_1) \to \mathsf{L}(\mathsf{N}, l_2)$$
$$\Sigma_{IS}(\mathtt{ins}) = (\mathsf{N}, \mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N}) \qquad [\Sigma_{IS}]\,(\mathtt{ins}) = (\mathsf{N}, \mathsf{L}(\mathsf{N}, l_3)\,, x_2) \to \mathsf{L}(\mathsf{N}, l_4)$$

Note that at this stage we do not imply that $l_1 = l_2$ or $l_1 = l_3$ as these will become naturally a part of the constraints gathered by the type derivation of the program's code.

We denote resource parameters of lists and sums by l_i and s_i respectively and resource parameters of functions by x_i like we did in Section 1, though this naming convention is of course completely arbitrary and merely a matter of human convenience. (To complete the naming convention of resource variables ranging over $\mathbb{N}$, we denote unknown resource variables of contexts by n or m_i.)

Lemma 2.4.9. *For all zero-order* LF*-types* A *it holds that any instance of* $[A]$ *is a pure zero-order* $\widehat{\mathsf{LFPL}}_R$ *type.*

Proof. Trivial, by induction on the composition of LF-types. □

Lemma 2.4.10. *Let* Σ *be a* LF*-signature.*

$$\forall \hat{\Sigma} \in \widehat{\mathsf{LFPL}}_R.\left|\hat{\Sigma}\right| = \Sigma \implies \hat{\Sigma} \text{ is an instance of } [\Sigma]$$

Proof. We will prove the slightly more general version

$$\forall A \in \mathsf{LF}\text{-types}.\forall \hat{A} \in \widehat{\mathsf{LFPL}}_R\text{-types}.\left|\hat{A}\right| = A \implies \hat{A} \text{ is an instance of } [A]$$

by induction on the composition of LF-types:

1, N: Trivial, as the type-schemes of these are constant.

$\mathsf{L}(A)$**:** By appeal to the induction hypothesis, the only $\widehat{\mathsf{LFPL}}_R$-types mapped by $|\cdot|$ to $\mathsf{L}(A)$ are $\mathsf{L}([A], n)$ for $n \in \mathbb{N}$, and all are clearly instances of the $\widehat{\mathsf{LFPL}}_R$-typescheme $\mathsf{L}([A], x)$.

$(A_1, \ldots, A_n) \to B$**:** Any $\widehat{\mathsf{LFPL}}$ preimage of a LF first-order type under $|\cdot|$ is of the form $(A'_1, \ldots, A'_n, k) \to B'$ for $n, k \in \mathbb{N}$. By definition $[(A_1, \ldots, A_n) \to B] = \big([A_1], \ldots, [A_n], x\big) \to [B]$ and so by the use of the induction hypothesis on $A_1, \ldots, A_n, B$ we are finished.

The remaining cases are similar. □

We intend to collect all the inequalities arising along a program's type derivation. As the essential core of each LF typing rule is identical to that of the corresponding $\widehat{\mathsf{LFPL}}$ typing rule, this turns out to be a successful approach. But before we can do this we must ensure that this will be a well-defined definition of $(\!|\cdot|\!)$, hence we have to show that a $\widehat{\mathsf{LFPL}}$ (or LF) type derivation is deterministic.

Lemma 2.4.11. *Let $P \in \widehat{\mathsf{LFPL}}_R$. Assume that Σ_1 and Σ_2 are two signatures that differ only on some of the resources numbers of the contained $\widehat{\mathsf{LFPL}}_R$ types.*

Then all valid $\widehat{\mathsf{LFPL}}_R$ type derivations for P under Σ_1 and Σ_2 have the same structure, i.e. differ at most by the values of the resource numbers.

Proof. By induction on the compositions of terms: With the exceptions of sums, there is only one $\widehat{\mathsf{LFPL}}$ type rule for each term constructor. So the outermost term constructor uniquely determines the type rule to be applied next to that term. So if Σ_1 is a valid signature and Σ_2 is not, then the type derivation under Σ_2 can only fail due to a violation of a side-condition.

The order in which the type derivations for the sub-terms in the premises of a type rule are derived does not matter, as the type derivations of sub-terms do not interfere with each other. In the case of sums, the problem of deciding whether variant I or II of a type rule is appropriate is non-existent in $\widehat{\mathsf{LFPL}}_R$, as we decided to forfeit the type $R + P$ in $\widehat{\mathsf{LFPL}}_R$. □

Example 2.4.12. Note that it is possible to obtain two different $\widehat{\mathsf{LFPL}}_R$ type derivations even for one and the same signature. Consider a program fragment like

$$\mathsf{match\ nil\ with\ |nil} \Rightarrow e_1\ \mathsf{|cons}(h, t) \Rightarrow e_2$$

The type rules allow us to derive a multitude of types for the term nil: the types $\mathsf{L}(A, 0), \mathsf{L}(B, 1), \mathsf{L}(C, 2), \ldots$ are all possible (but the possible use of h or t within the

term e_2 may restrict the type). This may affect the allowed values of the resource numbers, since a type derivation may even fail when we choose the wrong type (i.e. not enough resources), as the term e_2 may only be typable with plenty of resource available, though we know for sure that e_2 will *never* be executed.

This problem is quite artificial and not of relevance for the further results. Nevertheless it is worth to note that constructs like the one just presented will cause additional resource variables to appear within $(\!|P|\!)$ which are not contained in a program's signature: When building the skeleton of the type derivation of the program fragment written above, nil will be assigned the type $\mathsf{L}(A, l_j)$. Within the constraints there will be inequalities ensuring that l_j is at least large enough to satisfy the resources consumed by a hypothetic (but never happening) computation of e_2, but there will be no upper bounds on l_j whatsoever. With respect to Observation 2.4.16, variables like l_j can only be eliminated by removing such senseless constructs from the program's code.

Definition 2.4.13 (The ILP $(\!|P|\!)$)**.** For $P \in \mathsf{LF}$ with signature Σ define $(\!|P|\!)$ to be the integer linear program obtained by gathering all the inequalities arising when typing P under signature $[\Sigma]$ with the typing rules for $\widehat{\mathsf{LFPL}}_R$. Any new variable needed is always assumed to be fresh. Furthermore add assertions that all occurring variables are non-negative. By Lemma 2.4.11 this is well-defined, as the structure of the $\widehat{\mathsf{LFPL}}_R$ type derivation determines the set of inequalities.

We leave the objective function for a later discussion and define it for now to be constant.

Observation 2.4.14. Observe that when dealing with $\widehat{\mathsf{LFPL}}$ type-schemes some $\widehat{\mathsf{LFPL}}$ typing rules like Function-Application, Variable and even List-Construction require type unification in some cases. However, this type unification is unproblematic and always simple to solve as the $\widehat{\mathsf{LFPL}}$ type-schemes are only allowed to differ by the contained integer variable names, hence the unification only requires some simple equalities among pairs of the integer variables occurring in $[\Sigma]$.

We could explicitly add these equations to the typing rules[11], but they are unnecessary when we are dealing with known integers and would only obscure the important parts. So whenever dealing with $\widehat{\mathsf{LFPL}}$ type-schemes we assume these equations to be implicitly added. The following instructive example shall sufficiently describe the necessary amendments to be added to each LFPL type-rule, and so we refrain from stating the rules again for type-schemes.

[11]This could be simplified by the introduction of a recursive function returning the necessary equations to unify two type-schemes

Example 2.4.15. We may derive

$$\frac{\begin{array}{c}\Gamma_1, 12 \vdash_\Sigma e_h : \mathsf{N} + (\mathsf{N}, 4) \\ \Gamma_2, 13 \vdash_\Sigma e_t : \mathsf{L}\big(\mathsf{N} + (\mathsf{N}, 4), 5\big)\end{array}}{\Gamma_1, \Gamma_2, 42 \vdash_\Sigma \mathsf{cons}(e_h, e_t) : \mathsf{L}\big(\mathsf{N} + (\mathsf{N}, 4), 5\big)}\,(42 \geq 12 + 13 + (1 + 5))$$

but we need a rule like

$$\frac{\begin{array}{c}\Gamma_1, m_1 \vdash_\Sigma e_h : \mathsf{N} + (\mathsf{N}, s_1) \\ \Gamma_2, m_2 \vdash_\Sigma e_t : \mathsf{L}\big(\mathsf{N} + (\mathsf{N}, s_2), l_1\big)\end{array}}{\Gamma_1, \Gamma_2, n \vdash_\Sigma \mathsf{cons}(e_h, e_t) : \mathsf{L}\big(\mathsf{N} + (\mathsf{N}, s_3), l_2\big)}\begin{pmatrix} n \geq m_1 + m_2 + (1 + l_2) \\ s_1 = s_2 = s_3 \\ l_1 = l_2 \end{pmatrix}$$

in order to deal correctly with the $\widehat{\mathsf{LFPL}}$ type-scheme $\mathsf{L}\big(\mathsf{N} + (\mathsf{N}, x), y\big)$ within type derivations.

Observation 2.4.16. In the example above at least m_1 and m_2 would be fresh variables to be added,[12] but note that fresh variables of this particular kind can always be eliminated. Let us explain this at the example above for the freshly introduced variable m_2. Note that in all type rules there is a one-to-one correspondence between the occurring m_i and the sub-terms, so we can concentrate on the sub-term e_t in the given example. We have to distinguish two possibilities for e_t now:

e_t **contains no sub-terms:** Then the statement $\Gamma_2, m_2 \vdash_\Sigma e_t : \mathsf{L}(A, l_1)$ must be proved (if at all provable) by a terminating type rule. As already observed in the proof of Lemma 2.2.2, terminating type rules impose no further restriction on the integer variable connected to the context. Hence we can set the value of m_2 safely to zero, as m_2 only appeared on the right-hand side of the inequality (or inequalities) of the preceding rule.

e_t **contains sub-terms:** Now m_2 appears for the first time on the left-hand side of the inequalities for the type rule applied to e_t. Note that each left-hand side always consists of only one variable.

- Suppose that the outermost term constructor of e_t is a non-branching instruction. Then we have only one inequality of the form $m_2 \geq W$, where W stands for an arbitrary integer expression. As we are merely interested in a program's signature, we can replace all the previous occurrences of m_2 by W, which are all right-hand side occurrences as previously noted, thus eliminating m_2.

[12] e.g. for $e_t = \mathsf{nil}$ the variables s_2, l_1 would also be fresh

- Otherwise assume that the outermost term constructor of e_t is a branching operation, hence we get two inequalities $m_2 \geq W_1$ and $m_2 \geq W_2$. Since we cannot predict which of those two inequalities will be sharp, we have to copy each of the inequalities where m_2 occurs and replace m_2 with W_1 and W_2 respectively within the original and the copy. Note that this may cause an exponential blow up of the number of inequalities, since eliminating variables contained in both W_1 and W_2 requires further copying of *all* the just copied inequalities again. In this case m_2 functions in the same manner as the fresh variables introduced to prevent the exponential blow up of the number of inequalities in the construction of $(\!|P|\!)$ in Section 1. Nevertheless m_2 *could* be eliminated again.

So finally we are left with an integer linear program precisely over the variables of $[\Sigma]$, whose values are all that is required in order to determine the $\widehat{\mathsf{LFPL}}_R$ signature of a LF program.

It shall also be mentioned at this point that we could have turned all inequalities of the typing rule constraints to equations. The m_i would automatically serve as the required slack-variables. So what we have just described here is an elimination of slack-variables, as it is possible when remaining with the inequalities as we have chosen.

Finally note that the elimination is by no means necessary for the following results, so the exponential blow up of the set of inequalities can be easily avoided by keeping all additional variables. The integer linear program can still be solved within polynomial time, as shown soon! However we will use this elimination technique in the presented examples for the sake of simplicity.

Now we are ready to state and prove the main theorem of this section:

Theorem 2.4.17. *Let P be any* LF*-program with signature* Σ*.*

$$\exists \hat{\Sigma} \in \widehat{\mathsf{LFPL}}_R . \left|\hat{\Sigma}\right| = \Sigma \wedge P \text{ is a well-typed } \widehat{\mathsf{LFPL}}_R\text{-program under } \hat{\Sigma}$$
$$\Longleftrightarrow$$
$$(\!|P|\!) \text{ is solvable}$$

Proof.

$\Longleftarrow$ Assume that $(\!|P|\!)$ is solvable and let η be a solution. Now P must be well-typed under $[\Sigma]_\eta$ as the core of the typing rules for LF and $\widehat{\mathsf{LFPL}}_R$ are identical except for the constraints added to each typing rule, but η must solve all these constraints by the construction of $(\!|P|\!)$.

$\Longrightarrow$ $\hat{\Sigma}$ must be an instance of $[\Sigma]$ by Lemma 2.4.10. Hence the set of inequalities arising as side-conditions of a type derivation of P under $\hat{\Sigma}$ is equal to the set of inequalities of $(\!|P|\!)$ by Lemma 2.4.11. Thus we derive a valid instantiation of the variables of $(\!|P|\!)$.

□

Thus we have accomplished our main goal and reduced the LF/LFPL-translation problem to the problem of solving an integer linear program. First we have convinced ourselves that translating to $\widehat{\mathsf{LFPL}}$ is equally useful for our purposes as the translation from $\widehat{\mathsf{LFPL}}$ to LFPL is trivial and that the languages are of equal expressiveness. Then we have seen that the task of finding a translation to $\widehat{\mathsf{LFPL}}$ reduces to the task of finding a valid $\widehat{\mathsf{LFPL}}$ signature and that for the sake of feasibility we must further restrict to the fragment $\widehat{\mathsf{LFPL}}_R$, as without further limitations on the scope of accepted LF-programs solving this question is provably $\mathcal{NP}$-complete. Once a solution to this integer linear program is known, constructing the $\widehat{\mathsf{LFPL}}_R$-signature is trivial, as we have shown that the solution induces a valid instantiation of the enriched LF-signature yielding the $\widehat{\mathsf{LFPL}}_R$-signature. So we finally face now the question of solving the constructed integer linear program.

The feasibility of $(\!|P|\!)$

It is well-known that the problem of solving an arbitrary integer linear program belongs to the complexity class $\mathcal{NP}$. Even answering the question whether there exists an integral solution at all is already known to be $\mathcal{NP}$-complete (e.g. see [Sch86]).

However it is still possible to show feasibility for some classes of integer linear programs as we did for $(\!|P|\!)$ in Section 1: In the proof of Theorem 1.4 we showed that the polyhedron described by the relaxed rational linear program is integral, i.e. that all vertices of the polyhedron are integral. Since it is always feasibly possible to find an optimal rational solution among the polyhedron's vertices, solving the integer linear program reduces then to solving the relaxed rational linear program.

Alas, this time we cannot hope to prove that the polyhedron described by the relaxed rational linear program derived from an arbitrary LF-program is integral, as we can provide a counterexample.

Example 2.4.18. Let $P \in \mathsf{LF}$ with signature Σ be defined as:

$$\Sigma(\mathtt{tpo}) = (\mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N})$$

$$\begin{aligned}
\mathtt{tpo}(l) = \mathsf{match}\ & l\ \mathsf{with} \\
& \mid \mathsf{nil} \qquad\qquad \Rightarrow \mathsf{nil} \\
& \mid \mathsf{cons}(h_1, t_1) \Rightarrow \mathsf{match}\ t_1\ \mathsf{with} \\
& \qquad \mid \mathsf{nil} \qquad\qquad \Rightarrow \mathsf{cons}(h_1, \mathsf{nil}) \\
& \qquad \mid \mathsf{cons}(h_2, t_2) \Rightarrow \mathsf{cons}\Big(h_1, \mathsf{cons}\big(h_2, \mathsf{cons}(h_1 + h_2, \mathtt{tpo}(t_2))\big)\Big)
\end{aligned}$$

For an example we have

$$\begin{aligned}
&\mathtt{tpo}\big([7,5,2,2,7]\big) \\
&\qquad = [7,5,12] +\!\!+\, \mathtt{tpo}\big([2,2,7]\big) = [7,5,12,2,2,4] +\!\!+\, \mathtt{tpo}\big([7]\big) \\
&\qquad\qquad = [7,5,12,2,2,4,7]
\end{aligned}$$

where the symbol $+\!\!+$ stands for list concatenation.

Assume further that enriching the signature yields

$$[\Sigma]\,(\mathtt{tpo}) = (\mathsf{L}(\mathsf{N}, l_2)\,, x_1) \longrightarrow \mathsf{L}(\mathsf{N}, l_3)$$

So we seek the type derivation for

$$\big\{l : \mathsf{L}(\mathsf{N}, l_2)\big\}, x_1 \ \vdash_\Sigma \big(\textit{the defining body of } \mathtt{tpo}(l)\big) : \mathsf{L}(\mathsf{N}, l_3)$$

which finally gives us

$$(\!|P|\!) = \left\{ \begin{array}{l} x_1 \geq 0 \\ x_1 \geq -\ (1 + l_2) + \ \ (1 + l_3) \\ x_1 \geq -2(1 + l_2) + 3(1 + l_3) + x_1 \end{array} \right\} \text{ with } \begin{pmatrix} x_1 \\ l_2 \\ l_3 \end{pmatrix} \in \mathbb{N}^3$$

Note that it is easy to verify the derived inequalities for this short and simple function: Each of the inequalities corresponds to one possible branch of computation. For example the second inequality corresponds to the branch ending with the instruction $\mathsf{cons}(h_1, \mathsf{nil})$. This operation 'costs' $(1 + l_3)$ resources while we have already 'earned' $(1 + l_2)$ resources along that particular computational branch from successfully detaching the head of the input list. Now the number of the initial resources x_1 must be at least equal or greater than the overall cost of computing this branch.

Obviously the polyhedron described by these inequalities (and the additional restrictions that all variables are non-negative) has a non-integer vertex: $\chi = \left(0 \ \frac{1}{2} \ 0\right)^t$. which would result in the (invalid) signature

$$[\Sigma]\,(\mathtt{tpo})_\chi = \left(\mathsf{L}\left(\mathsf{N}, \tfrac{1}{2}\right), 0\right) \longrightarrow \mathsf{L}(\mathsf{N}, 0)$$

that could possibly mean that only every other list-node contains an extra resource.

Hence we cannot hope to prove a similar result like Theorem 1.4 in this section. One could object that it is possible to accept this certain fractional solution and change the size of the portion of heap-space each resource element represents to one-half its former size. Martin Hofmann already discussed in his work [Hof00] the possibility of each resource type element to correspond to a smaller heap-space portion in order to fit the slightly different space usage of a list-node compared to a tree-node (or elements of other dynamical structures), but concluded that is seemed more rewarding to deal with just one resource type that is large enough to hold a node of any used data structure as the space wastage is always within a constant multiple of the size of the data. However note that this case is even different, as we might obtain rationals that would require heap-space portions that are even smaller than the space units the hardware might be able to address.

Therefore it seems more sensible to insist on integral solutions to the linear program constructed. Of course in this case we can luckily obtain a solution by rounding up, but this is not always possible as the next example quickly shows.

Example 2.4.19. Extend $P \in \mathsf{LF}$ with signature Σ from the previous Example 2.4.18 by adding two other functions `tos` and `sec` as follows:

$$\begin{aligned}
\Sigma(\mathtt{tos}) &= (\mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N}) \\
\Sigma(\mathtt{sec}) &= (\mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N})
\end{aligned}$$

$$\begin{aligned}
[\Sigma]\,(\mathtt{tos}) &= (\mathsf{L}(\mathsf{N}, l_1), x_0) \longrightarrow \mathsf{L}(\mathsf{N}, l_3) \\
[\Sigma]\,(\mathtt{sec}) &= (\mathsf{L}(\mathsf{N}, l_1), x_2) \longrightarrow \mathsf{L}(\mathsf{N}, l_2)
\end{aligned}$$

For convenience we have already unified some variables as mentioned in Observation 2.4.14, due to the definition of `tos` as `tpo` $\circ$ `sec` to come:

$$
\begin{aligned}
&\texttt{tos}(l) = \texttt{tpo}(\texttt{sec}(l))\\
&\texttt{sec}(l) = \mathsf{match}\ l\ \mathsf{with}\\
&\qquad \mid \mathsf{nil} \qquad\qquad \Rightarrow \mathsf{nil}\\
&\qquad \mid \mathsf{cons}(h_1, t_1) \Rightarrow \mathsf{match}\ t_1\ \mathsf{with}\\
&\qquad\qquad \mid \mathsf{nil} \qquad\qquad \Rightarrow \mathsf{cons}(h_1, \mathsf{nil})\\
&\qquad\qquad \mid \mathsf{cons}(h_2, t_2) \Rightarrow \mathsf{match}\ t_2\ \mathsf{with}\\
&\qquad\qquad\qquad \mid \mathsf{nil} \qquad\qquad \Rightarrow \mathsf{cons}\big(h_1, \mathsf{cons}(h_2, \mathsf{nil})\big)\\
&\qquad\qquad\qquad \mid \mathsf{cons}(h_3, t_3) \Rightarrow \mathsf{cons}\big(h_1, \mathsf{cons}(h_2, \texttt{sec}(t_3))\big)
\end{aligned}
$$

Illustrating function `sec` we see

$$
\begin{aligned}
&\texttt{sec}\big([7,5,12,2,2,2,4,7]\big)\\
&\qquad = [7,5] +\!\!+ \texttt{sec}\big([2,2,4,7]\big) = [7,5,2,2] +\!\!+ \texttt{sec}\big([7]\big)\\
&\qquad\qquad = [7,5,2,2,7]
\end{aligned}
$$

therefore `tos` replaces the third list element by the sum of its two predecessors.

After eliminating the intermediate variables – as stated in Observation 2.4.16 – we finally obtain

$$
(\!|P|\!) = \left\{ \begin{array}{l}
x_0 \geq x_1 + x_2\\
x_1 \geq 0\\
x_1 \geq -\ (1+l_2) +\ \ (1+l_3)\\
x_1 \geq -2(1+l_2) + 3(1+l_3) + x_1\\
x_2 \geq 0\\
x_2 \geq -\ (1+l_1) +\ \ (1+l_2)\\
x_2 \geq -2(1+l_1) + 2(1+l_2)\\
x_2 \geq -3(1+l_1) + 2(1+l_2) + x_2
\end{array} \right\} \text{ with } \begin{pmatrix} x_0\\ x_1\\ x_2\\ l_1\\ l_2\\ l_3 \end{pmatrix} \in \mathbb{N}^6
$$

The non-integer (non-negative) solution to the relaxed rational linear program being: $\begin{pmatrix} 1 & 0 & 1 & 0 & \frac{1}{2} & 0 \end{pmatrix}^t$, which is a vertex of the set of solutions and that in addition cannot be rounded up to a valid solution to the integer linear program we want to solve.

Nevertheless we claim and prove:

Theorem 2.4.20. *Let $P \in \mathsf{LF}$ with signature Σ. A solution for $(\!|P|\!)$ is constructible in polynomial time, if $(\!|P|\!)$ is solvable at all.*

Proof. Assume that $(\!|P|\!)$ is an integer linear program over n variables. Relax $(\!|P|\!)$ to a linear program and let $\chi \in \left(\mathbb{R}^+\right)^n$ denote a solution for it, obtained in polynomial time using a standard algorithm. Obviously $\chi \in \left(\mathbb{Q}^+\right)^n$ as $(\!|P|\!)$ is rational.

Please observe that there are only four type rules involving an integer constant, namely the rules List-Construction, List-Elimination, Tree-Construction and Tree-Elimination. Note that variables connected either to list- or tree-types always occur together with those integer constants, which is in both cases one. So we replace each occurrence of $(1 + \chi_i)$ by χ'_i, adding the inequality $\chi'_i \geq 1$ and thus obtaining a new integer linear program, named $(\!|P|\!)'$ for reference. Notice that $(\!|P|\!)'$ is a homogeneous linear program, except for those inequalities just added asserting that some variables have to be equal or greater or than one. Let χ' denote the solution for $(\!|P|\!)'$ derived from χ. Multiply χ' by the least common multiple of all occurring denominators to obtain $\widehat{\chi}' \in \mathbb{N}^n$. Now $\widehat{\chi}'$ is a solution for $(\!|P|\!)'$ as well, since multiplying χ' by a natural number greater than one does neither affect the validity of the homogeneous inequalities nor the validity of the inequalities of the type $\chi'_i \geq 1$. Therefore we can also derive $\widehat{\chi} \in \mathbb{N}^n$ as a solution for $(\!|P|\!)$ by subtracting one from the values of the variables of $\widehat{\chi}'$ that are connected to list- or tree-types. □

Since $[\Sigma]$ as given in Definition 2.4.7 and $(\!|P|\!)$ as described in Definition 2.4.13 can be constructed within polynomial time and by Theorem 2.4.17 and Theorem 2.4.20 we deduce:

Corollary 2.4.21. *Let P be a LF-program with signature Σ. Finding a $\widehat{\mathsf{LFPL}}_R$-signature $\hat{\Sigma}$ with $\left|\hat{\Sigma}\right| = \Sigma$ so that P is a well-typed $\widehat{\mathsf{LFPL}}_R$-program can be done within polynomial time, if such a signature exists at all.*

Now we will apply the procedure described in the proof of Theorem 2.4.20 to Example 2.4.19:

Example (**2.4.19**, revisited)**.** Respecting the naming conventions in the proof, we obtain the 'almost homogeneous' system:

$$(\!|P|\!)' = \left\{ \begin{array}{l} x_0 \geq x_1 + x_2 \\ x_1 \geq 0 \\ x_1 \geq -\ l'_2 + \ l'_3 \\ x_1 \geq -2l'_2 + 3l'_3 + x_1 \\ x_2 \geq 0 \\ x_2 \geq -\ l'_1 + \ l'_2 \\ x_2 \geq -2l'_1 + 2l'_2 \\ x_2 \geq -3l'_1 + 2l'_2 + x_2 \\ l'_1 \geq 1 \\ l'_2 \geq 1 \\ l'_3 \geq 1 \end{array} \right\} \text{ with } \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ l'_1 \\ l'_2 \\ l'_3 \end{pmatrix} \in \mathbb{N}^6$$

and the solutions to each step as depicted in the proof

$$\chi = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ \frac{1}{2} \\ 0 \end{pmatrix} \qquad \chi' = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ \frac{3}{2} \\ 1 \end{pmatrix} \qquad \widehat{\chi}' = \begin{pmatrix} 2 \\ 0 \\ 2 \\ 2 \\ 3 \\ 2 \end{pmatrix} \qquad \widehat{\chi} = \begin{pmatrix} 2 \\ 0 \\ 2 \\ 1 \\ 2 \\ 1 \end{pmatrix}$$

Reassuring oneself that $\widehat{\chi}$ is indeed a solution for $(\!|P|\!)$ is left to the reader, we simply state the valid computed $\widehat{\mathsf{LFPL}}_R$ signature:

$$[\Sigma]\,(\mathtt{tos})_{\widehat{\chi}} = (\mathsf{L}(\mathsf{N},1)\,,2) \longrightarrow \mathsf{L}(\mathsf{N},1)$$
$$[\Sigma]\,(\mathtt{sec})_{\widehat{\chi}} = (\mathsf{L}(\mathsf{N},1)\,,2) \longrightarrow \mathsf{L}(\mathsf{N},2)$$
$$[\Sigma]\,(\mathtt{tpo})_{\widehat{\chi}} = (\mathsf{L}(\mathsf{N},2)\,,0) \longrightarrow \mathsf{L}(\mathsf{N},1)$$

Choosing the objective function for $(\!|P|\!)$

We have now proved that it is possible to construct at least *one* solution to the integer linear program $(\!|P|\!)$ derived from an arbitrary LF-program P, but we have not discussed the quality of the solution obtained. Our method of constructing the integer solution does not involve the objective function of $(\!|P|\!)$, except for constructing the initial non-integer solution. We might lose minimality at the end when multiplying with the least common multiple.

However, we do not consider this as an important defect, because it seems rather unclear how the objective function should be defined to capture our notion of optimality, as a further exploration of the previous example reveals:

Example (**2.4.19**, exerted once more)**.** In order to discuss the quality of a solution of $(\!|P|\!)$, we first state all the solutions of the relaxed linear program by the complete polyhedron given by points in homogeneous coordinates,[13] the points being written in rows and the first column corresponding to the added dimension.

The polyhedron was computed with aid of the free software package *polymake*, see [G&J00].

$$\left\{\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 1/2 & 0 \\ 0 & 1 & 0 & 1 & 1 & 3/2 & 1 \\ 0 & 1 & 0 & 1 & 1 & 3/2 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 3/2 & 3/2 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}\right\}$$

Defining the objective function to minimise the sum of all variables on the left-hand side of a first-order type, – like we did in Section 1 – would obviously result in the solution $\check{\chi} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}^t$ (as a conic combination of the 7^{th} and the 9^{th} row). However, this solution might be unsatisfying: compare the $\widehat{\mathsf{LFPL}}$-signatures of `tos` for $\widehat{\chi}$ and $\check{\chi}$:

$$[\Sigma]_{\widehat{\chi}}(\mathtt{tos}) = (\mathsf{L}(\mathsf{N},1),2) \longrightarrow \mathsf{L}(\mathsf{N},1)$$
$$[\Sigma]_{\check{\chi}}(\mathtt{tos}) = (\mathsf{L}(\mathsf{N},1),0) \longrightarrow \mathsf{L}(\mathsf{N},0)$$

In the first case a call to function `tos` might cost two additional resources, but it also preserves all the resources contained within the list, which are many more resources in general. Additionally, preserving the type of the list is necessary for recursive applications of the function to lists. Nevertheless the second case might still be of better use in a scenario that applies `tos` to many lists having altogether an average length of two only.

[13] The polyhedron is obtained as the projection of the conic hull of the given set of points to the hyperplane with normal vector $(1\ 0\ \cdots\ 0)^t$ and also containing $(1\ 0\ \cdots\ 0)^t$, i.e. embedding the n-dimensional polyhedron in $(n+1)$-dimensional space with the property that each point belonging to the polyhedron has a coordinate of the form $(1\ \alpha\ \beta\ \cdots\ \zeta)^t$, hence in this example every conic combination of rows having the form $(1\ x_0\ x_1\ x_2\ l_1\ l_2\ l_3)^t$ belongs to the set of solutions.

So we conclude that the notion of optimality rather depends on the environment in which a particular function is used in and therefore we see no point in a general discussion of this problem now. Furthermore Martin Hofmann pointed out that that constructing an 'optimal' signature according to a given objective function is a $\mathcal{NP}$-hard problem:

Theorem 2.4.22. *Let P be a LF-program with signature Σ. Finding a $\widehat{\mathsf{LFPL}}_R$-signature $\hat{\Sigma}$ with $\left|\hat{\Sigma}\right| = \Sigma$ so that P is a well-typed $\widehat{\mathsf{LFPL}}_R$-program under $\hat{\Sigma}$* and additionally *that $\hat{\Sigma}$ is optimal with respect to a given objective function on the variables of $[\Sigma]$ is an $\mathcal{NP}$-hard task.*

Proof. Again by reducing 3SAT. Let $\Phi = (u_{11} \vee u_{12} \vee u_{13}) \wedge \cdots \wedge (u_{n1} \vee u_{n2} \vee u_{n3})$ be an instance of 3SAT. Construct P_Φ and Σ_Φ now in the following way:

- Suppose that there are m different variables $v_k \in \Phi$, define for each

$$\begin{aligned} f_k(*) &:= f_k(*) & \bar{f}_k(*) &:= \bar{f}_k(*) \\ \Sigma_\Phi(f_k) &:= 1 \to \mathsf{L}(1) & \Sigma_\Phi(\bar{f}_k) &:= 1 \to \mathsf{L}(1) \end{aligned}$$

$$h_k(v) := h_k\left(h_k\left(\begin{array}{c}\mathsf{match}\ f_k(*)\ \mathsf{with}\ |\mathsf{nil} \Rightarrow \mathsf{nil}\ |\mathsf{cons}(h_1,t_1) \Rightarrow \\ \mathsf{match}\ \bar{f}_k(*)\ \mathsf{with}\ |\mathsf{nil} \Rightarrow \mathsf{nil}\ |\mathsf{cons}(h_2,t_2) \Rightarrow \\ \mathsf{cons}\Big(*, \mathsf{cons}\big(*, \mathsf{cons}(*, \mathsf{nil})\big)\Big)\end{array}\right)\right)$$

$$\Sigma_\Phi(h_k) := \mathsf{L}(1) \to \mathsf{L}(1)$$

- For each of the n clauses in Φ define

$$g_i(v) := g_i\left(g_i\left(cu_{i1}\left[w \Big\backslash cu_{i2}\left[w \Big\backslash cu_{i3}\left[w \Big\backslash \mathsf{cons}(*,\mathsf{cons}(*,\mathsf{cons}(*,\mathsf{cons}(*,\mathsf{nil}))))\right]\right]\right]\right)\right)$$

$$\Sigma_\Phi(g_i) := \mathsf{L}(1) \to \mathsf{L}(1)$$

where

$$cu_{ij} := \begin{cases} \mathsf{match}\ f_k(*)\ \mathsf{with}\ |\mathsf{nil} \to \mathsf{nil}\ |\mathsf{cons}(h_j,t_j) \to w & \text{if } u_{ij} = v_k \\ \mathsf{match}\ \bar{f}_k(*)\ \mathsf{with}\ |\mathsf{nil} \to \mathsf{nil}\ |\mathsf{cons}(h_j,t_j) \to w & \text{if } u_{ij} = \neg v_k \end{cases}$$

We illustrate the construction again by considering the function g_1 corresponding to the clause

$$v_1 \vee \neg v_2 \vee v_3$$

$$g_1(v) := g_1\left(\begin{array}{l} g_1\left(\begin{array}{l} \mathsf{match}\ f_1(*)\ \mathsf{with}\ |\mathsf{nil} \Rightarrow \mathsf{nil}\ |\mathsf{cons}(h_1,t_1) \Rightarrow \\ \quad \mathsf{match}\ \bar{f}_2(*)\ \mathsf{with}\ |\mathsf{nil} \Rightarrow \mathsf{nil}\ |\mathsf{cons}(h_2,t_2) \Rightarrow \\ \quad\quad \mathsf{match}\ f_3(*)\ \mathsf{with}\ |\mathsf{nil} \Rightarrow \mathsf{nil}\ |\mathsf{cons}(h_3,t_3) \Rightarrow \\ \quad\quad\quad \mathsf{cons}\big(*, \mathsf{cons}\big(*, \mathsf{cons}(*, \mathsf{cons}(*, \mathsf{nil}))\big)\big) \end{array}\right) \end{array}\right)$$

hence again each h_k represents the clause $v_k \vee \neg v_k$. Therefore, if

$$[\Sigma_\Phi]\,(f_k) = (1, x_k) \multimap \mathsf{L}(1, l_k) \qquad\qquad [\Sigma_\Phi]\,(\bar{f}_k) = (1, \bar{x}_k) \multimap \mathsf{L}\big(1, \bar{l}_k\big)$$

the inequality $l_k + \bar{l}_k \geq 1$ must hold for any instance $\hat{\Sigma}_\Phi$ of $[\Sigma_\Phi]$ that allows a valid $\widehat{\mathsf{LFPL}}_R$ type derivation for P_Φ: since there is one computational branch of h_k which consumes at least 3 resources, these must be obtained by matching f_k and $\bar{f}_k$, because $x_k, \bar{x}_k$ are forced to be zero due to the recursive calls. Now the two match operations deliver only two resources from the detached list-nodes, hence either one of the lists must contain spare resources.

By the same reasoning for g_i follows that if $\hat{\Sigma}_\Phi$ additionally satisfies $l_k + \bar{l}_k = 1$ for each i, then

$$\rho(v_i) := \begin{cases} true & \hat{\Sigma}_\Phi(f_k) = (1,0) \multimap \mathsf{L}(1,1) \\ false & \hat{\Sigma}_\Phi(\bar{f}_k) = (1,0) \multimap \mathsf{L}(1,1) \end{cases}$$

is valid valuation that satisfies Φ.

So if we choose the objective function

$$f_{[\Sigma_\Phi]} := \sum_{k=1}^{m} l_k + \bar{l}_k$$

then

$$l_k + \bar{l}_k \geq 1 \implies f_{[\Sigma_\Phi]} \geq n \qquad \text{and hence} \qquad f_{[\Sigma_\Phi]} = n \implies l_k + \bar{l}_k = 1$$

Thus Φ is satisfiable if and only if there exits a $\widehat{\mathsf{LFPL}}_R$ signature $\hat{\Sigma}_\Phi$ such that P_Φ is well-typed under $\hat{\Sigma}_\Phi$ and $f_{[\Sigma_\Phi]}(\hat{\Sigma}_\Phi) = m$. □

We furthermore expect that choices for signatures are dramatically reduced when dealing with praxis-life linear functional programs, where the functions are interwoven due to mutual calls. Thus we regard it sufficient to provide at least one $\widehat{\mathsf{LFPL}}$ signature, whenever one exists at all.

We close Section 2 now by computing the $\widehat{\mathsf{LFPL}}$-signature of another program example: The computation of the Huffman-Coding tree as given in [B&W88, p.242].[14]

[14]Due to the absence of a library we have to merge some standard functions like `map` into other functions, but otherwise we try to stay as closely to the code presented in [B&W88] as possible when translating into a LF-style notation.

Example 2.4.23 (Huffman tree)**.** A well-known method of data compression is the prefix coding method, where the "key" consists of a tree bearing parts of the coded information at its leaf[15] and the actual "data" being given by a list of prefix codes, representing paths in the tree leading to one of the leaves. The idea is that parts that occur more often within the original data are stored at higher levels in the tree, thus being coded by shorter tree paths. An optimal method of generating this tree is known as the Huffman-Coding.

We suppose that we receive a list of pairs of integers, the first of the pair representing the part of data to be encoded as a unit within the tree and the second representing its frequency. In order to build the Huffman tree we will compute the following: First we sort the list of pairs by the frequencies (the second of the pair) and turn the first number of each pair (the data) into a tree with a single node holding the number. Then we will successively merge two trees of that list into one, until we are left with one single tree. As a subfunction we need the function `insert` which is essentially a slightly modified version of the function `ins` from Example 1.8 (Insertion-Sort).

Let $P \in \mathsf{LF}$ with signature Σ be defined as:

$$\begin{array}{lcccc}
\Sigma(\texttt{huffman}) & = & (\mathsf{L}(\mathsf{N} \otimes \mathsf{N})) & \longrightarrow & \mathsf{T}(\mathsf{N}) \\
\Sigma(\texttt{maptree}) & = & (\mathsf{L}(\mathsf{N} \otimes \mathsf{N})) & \longrightarrow & \mathsf{L}(\mathsf{T}(\mathsf{N}) \otimes \mathsf{N}) \\
\Sigma(\texttt{combine}) & = & (\mathsf{L}(\mathsf{T}(\mathsf{N}) \otimes \mathsf{N})) & \longrightarrow & \mathsf{T}(\mathsf{N}) \\
\Sigma(\texttt{insert}) & = & (\mathsf{T}(\mathsf{N}) \otimes \mathsf{N}, \mathsf{L}(\mathsf{T}(\mathsf{N}) \otimes \mathsf{N})) & \longrightarrow & \mathsf{L}(\mathsf{T}(\mathsf{N}) \otimes \mathsf{N})
\end{array}$$

[15] As we deal with trees having unlabelled leafs only the actual information must be stored in nodes having two leafs as their branches.

$$
\begin{aligned}
&\texttt{huffman}(l) = \texttt{combine}(\texttt{maptree}(l))\\
&\texttt{maptree}(l) = \mathsf{match}\ l\ \mathsf{with}\\
&\qquad\qquad \mid \mathsf{nil} \Rightarrow \mathsf{nil}\\
&\qquad\qquad \mid \mathsf{cons}(h,t) \Rightarrow \mathsf{match}\ h\ \mathsf{with}\ c \otimes w \Rightarrow\\
&\qquad\qquad\qquad \texttt{insert}\big(\mathsf{node}(c,\mathsf{leaf},\mathsf{leaf}) \otimes w, \texttt{maptree}(\mathtt{t})\big)\\
&\texttt{combine}(l) = \mathsf{match}\ l\ \mathsf{with}\\
&\qquad\qquad \mid \mathsf{nil} \Rightarrow \mathsf{leaf}\\
&\qquad\qquad \mid \mathsf{cons}(h_1,t_1) \Rightarrow \mathsf{match}\ h_1\ \mathsf{with}\ r_1 \otimes w_1 \Rightarrow\\
&\qquad\qquad\qquad \mathsf{match}\ t_1\ \mathsf{with}\\
&\qquad\qquad\qquad\qquad \mid \mathsf{nil} \Rightarrow r_1\\
&\qquad\qquad\qquad\qquad \mid \mathsf{cons}(h_2,t_2) \Rightarrow \mathsf{match}\ h_2\ \mathsf{with}\ r_2 \otimes w_2 \Rightarrow\\
&\qquad\qquad\qquad\qquad\qquad \texttt{combine}\big(\texttt{insert}(\mathsf{node}(0,r_1,r_2) \otimes (w_1+w_2), t_2)\big)\\
&\texttt{insert}(p,l) = \mathsf{match}\ l\ \mathsf{with} \Rightarrow\\
&\qquad\qquad \mid \mathsf{nil} \Rightarrow \mathsf{cons}(p,\mathsf{nil})\\
&\qquad\qquad \mid \mathsf{cons}(h,t) \Rightarrow\\
&\qquad\qquad\qquad \mathsf{match}\ p\ \mathsf{with}\ r_1 \otimes w_1 \Rightarrow\\
&\qquad\qquad\qquad \mathsf{match}\ h\ \mathsf{with}\ r_2 \otimes w_2 \Rightarrow\\
&\qquad\qquad\qquad \mathsf{if}\ w_1 \geq w_2\\
&\qquad\qquad\qquad\quad \mathsf{then}\ \mathsf{cons}\big(r_1 \otimes w_1, \mathsf{cons}(r_2 \otimes w_2, t)\big)\\
&\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{cons}\big(r_2 \otimes w_2, \texttt{insert}(r_2 \otimes w_2, t)\big)
\end{aligned}
$$

Let the enriched signature $[\Sigma]$ be as follows. Again, the necessary unifications due to the mutual use of list- and tree-types as described in Observation 2.4.14 have already been done, reducing the dimension from 17 to 7 and hence imposing many restrictions like we had predicted at the end of Section 2.4.

$$
\begin{array}{lcccc}
[\Sigma](\texttt{huffman}) & = & (\mathsf{L}(\mathsf{N}\otimes\mathsf{N}, l_1), x_1) & \longrightarrow & \mathsf{T}(\mathsf{N}, b_1)\\
[\Sigma](\texttt{maptree}) & = & (\mathsf{L}(\mathsf{N}\otimes\mathsf{N}, l_1), x_2) & \longrightarrow & \mathsf{L}(\mathsf{T}(\mathsf{N}, b_1)\otimes\mathsf{N}, l_2)\\
[\Sigma](\texttt{combine}) & = & (\mathsf{L}(\mathsf{T}(\mathsf{N}, b_1)\otimes\mathsf{N}, l_2), x_3) & \longrightarrow & \mathsf{T}(\mathsf{N}, b_1)\\
[\Sigma](\texttt{insert}) & = & (\mathsf{T}(\mathsf{N}, b_1)\otimes\mathsf{N}, \mathsf{L}(\mathsf{T}(\mathsf{N}, b_1)\otimes\mathsf{N}, l_2), x_4) & \longrightarrow & \mathsf{L}(\mathsf{T}(\mathsf{N}, b_1)\otimes\mathsf{N}, l_2)
\end{array}
$$

Leading us to the following integer linear program

$$(\!|P|\!) = \left\{ \begin{array}{l} x_1 \geq x_2 + x_3 \\ x_2 \geq 0 \\ x_2 \geq -(1+l_1) + x_4 + (1+b_1) + x_2 \\ x_3 \geq 0 \\ x_3 \geq -(1+l_2) \\ x_3 \geq -2(1+l_2) + x_3 + x_4 + (1+b_1) \\ x_4 \geq (1+l_2) \\ x_4 \geq (-1+2)(1+l_2) \\ x_4 \geq (-1+1)(1+l_2) + x_4 \end{array} \right\} \text{ with } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ l_1 \\ l_2 \\ b_1 \end{pmatrix} \in \mathbb{N}^8$$

The polyhedron of the relaxed linear program – given in homogeneous coordinates again – then is

$$\left\{ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \right\}$$

being already integral in this case. So we directly obtain the solution $(\,0\,0\,0\,1\,1\,0\,0\,)^t$ leading to the signature

$$\begin{array}{lcccl} [\Sigma]_\chi(\texttt{huffman}) & = & (\mathsf{L}(\mathsf{N} \otimes \mathsf{N}, 1), 0) & \to & \mathsf{T}(\mathsf{N}, 0) \\ [\Sigma]_\chi(\texttt{maptree}) & = & (\mathsf{L}(\mathsf{N} \otimes \mathsf{N}, 1), 0) & \to & \mathsf{L}(\mathsf{T}(\mathsf{N}, 0) \otimes \mathsf{N}, 0) \\ [\Sigma]_\chi(\texttt{combine}) & = & (\mathsf{L}(\mathsf{T}(\mathsf{N}, 0) \otimes \mathsf{N}, 0), 0) & \to & \mathsf{T}(\mathsf{N}, 0) \\ [\Sigma]_\chi(\texttt{insert}) & = & (\mathsf{T}(\mathsf{N}, 0) \otimes \mathsf{N}, \mathsf{L}(\mathsf{T}(\mathsf{N}, 0) \otimes \mathsf{N}, 0), 1) & \to & \mathsf{L}(\mathsf{T}(\mathsf{N}, 0) \otimes \mathsf{N}, 0) \end{array}$$

Note that this is also optimal (in any sense), as for a list with n nodes the tree to be computed must occupy exactly $2n - 1$ resources. One resource is wasted in the second computational branch of `combine` (the one ending with r_1). In this branch we have already computed the Huffman tree, but it is still stored in a list of length one. Destroying this list to reveal the tree gives us one resource (from the list-node), but we cannot return this resource due to our restriction to faithful signatures. One could save this resource of course by returning the singleton list instead of the tree, but this would require an awareness of the problem in LF — and furthermore the use of meta-reasoning enabling us to deal properly with such singleton lists.

3 On remaining problems and unventured ideas

Let us first recall all problems and open questions encountered so far, before we discuss any further:

Plotkins remark 1.3: Gordon Plotkin pointed out that we could change the part of Definition 1.2 regarding the standard integer infix $\star \in \{+, -, \times, \geq, \dots\}$ for elements of type N from

$$(\!|e_1 \star e_2|\!) = (\!|e_1|\!) + (\!|e_2|\!) \qquad \text{to} \qquad (\!|e_1 \star e_2|\!) = \max\big\{(\!|e_1|\!), (\!|e_2|\!)\big\}$$

since all the heap-space used during the computation of an element of a base type might be reclaimed immediately. It is not entirely clear if this idea can be generalised to all computations of elements of heap-free types,[16] and should therefore be further investigated.

Section 2.1: Is the expressive power of LFPL in terms of the set-theoretic semantic reduced by the restriction to programs having a faithful signature? Or in other words, does the expressive power of $\widehat{\mathsf{LFPL}}$ still equal exponential time? The answer seems to be no, but we have not really examined the problem yet! Are there any other disadvantages of faithful signatures that we might have missed noticing?

Section 2.4, Transforming LF to $\widehat{\mathsf{LFPL}}$: By proving that the $\widehat{\mathsf{LFPL}}$ signature problem is $\mathcal{NP}$-complete we discovered that it is still possible to 'borrow' resources from non-terminating program fragments like discussed at length in Section 2.1. Is it possible to banish such dangerous techniques from $\widehat{\mathsf{LFPL}}$ and is it even *sensible* to try to?

Section 2.4, Building the ILP for $\widehat{\mathsf{LFPL}}_R$: We showed that the $\widehat{\mathsf{LFPL}}$ signature problem is $\mathcal{NP}$-complete and therefore reduced to $\widehat{\mathsf{LFPL}}_R$. Are there any sensible heuristics that could help deciding which way round a sum-type should be set? Which sum-types should be swapped first when solving $(\!|P|\!)$ fails completely for the next try?

3.1 Eliminating the necessity for a linear typing

The linear typing of LFPL causes some inconvenience, as it is not native to the functional programming style. A linear typing is also completely unmotivated in a

[16] see forward to Section 3.1 for the definition "heap-free type"

language like LF, but it was our aim to be able to program in LF without giving thought to the use of heap-space resources, while comfortably receive the heap space usage by an automated computation. Therefore it would be nice to eliminate the restriction of linear typing.

Additionally, throughout most of the presented examples, we violated linearity by the multiple use of variables of the base types N and $\mathsf{1}$. We already mentioned this in Example 1.8 and promised to justify this later. Now we do this by referring to the contraction rule, which was already a part of the typing rules of LFPL invented by Martin Hofmann.

Contraction

$$\frac{\Gamma, x : A, y : A \overset{\mathsf{LFPL}}{\vdash_\Sigma} e : C}{\Gamma, x : A \overset{\mathsf{LFPL}}{\vdash_\Sigma} e_{\left[y \setminus x\right]} : C} \ (A \text{ is a heap-free type})$$

Where "A is a heap-free type" means that the type A does not contain the type constructors $\mathsf{L}(\cdot)\,, \mathsf{T}(\cdot)$ nor $\Diamond$, and hence can be copied without affecting the heap-space usage. Of course, the copy does require some space within the stack and has to be done by the interpreter or compiler as usual in implementations of functional languages.

Now the question arises how this rule can be transported to $\widehat{\mathsf{LFPL}}$ and whether or not it is possible to extend it further eventually eliminating the restriction to linear typing. But before we can try this, we need to formalise and generalise the property of types being heap-free.

Definition 3.1.1. Define

$$\langle\!\langle \cdot \rangle\!\rangle : \widehat{\mathsf{LFPL}}\text{-zero-order types} \longrightarrow \mathbb{N} \cup \infty$$

and say that $\langle\!\langle A \rangle\!\rangle$ is the *maximal content* of an $\widehat{\mathsf{LFPL}}$-type A. The intuition of $\langle\!\langle A \rangle\!\rangle$ is the number of resources that are at most contained in any element of type A. The defining equations are:

$$\begin{aligned} \langle\!\langle \mathsf{1} \rangle\!\rangle &= 0 & \qquad \langle\!\langle \mathsf{N} \rangle\!\rangle &= 0 \\ \langle\!\langle A \otimes B \rangle\!\rangle &= \langle\!\langle A \rangle\!\rangle + \langle\!\langle B \rangle\!\rangle & \qquad \langle\!\langle A + B \rangle\!\rangle &= \max\big(\langle\!\langle A \rangle\!\rangle\,, \langle\!\langle B \rangle\!\rangle\big) \\ \langle\!\langle \mathsf{L}(A) \rangle\!\rangle &= \infty & \qquad \langle\!\langle \mathsf{T}(A) \rangle\!\rangle &= \infty \\ \langle\!\langle (A, n) \rangle\!\rangle &= n + \langle\!\langle A \rangle\!\rangle \end{aligned}$$

A type A is called *heap-free* if and only if $\langle\!\langle A \rangle\!\rangle = 0$.

Now we can define the $\widehat{\mathsf{LFPL}}$ type rules

0-level Contraction

$$\frac{\Gamma, x:A, y:A, m \overset{\widehat{\mathsf{LFPL}}}{\vdash_\Sigma} e : C}{\Gamma, x:A, n \overset{\widehat{\mathsf{LFPL}}}{\vdash_\Sigma} e_{\left[y \backslash x\right]} : C} \left(n \geq m + \langle\!\langle A \rangle\!\rangle\right)$$

1-level Contraction (for lists)

$$\frac{\Gamma, x:\mathsf{L}(A,l_1), y:\mathsf{L}(A,l_1), m \overset{\widehat{\mathsf{LFPL}}}{\vdash_\Sigma} e : C}{\Gamma, x:\mathsf{L}(A,l_0), n \overset{\widehat{\mathsf{LFPL}}}{\vdash_\Sigma} e_{\left[y \backslash x\right]} : C} \begin{pmatrix} n \geq m \\ l_0 \geq 1 + 2l_1 + \langle\!\langle A \rangle\!\rangle \end{pmatrix}$$

which relieves us from rigid linearity: The 0-level Contraction allows the multiple use of $\widehat{\mathsf{LFPL}}$ types like $\mathsf{N}, \mathsf{N} \otimes \mathsf{N}$, etc. and even of non-heap-free types like $\mathsf{N} + (\mathsf{N}, 5)$, provided that there are enough resources available to construct another copy of it. The 1-level Contraction for lists then allows the implicit copying of lists of simpler types, provided that there are enough resources available to construct the required list-nodes. (Again, performing the copy-operation is the responsibility of the interpreter or compiler.) Since we do not know the length of the list, these resources must be contained within the list to be copied, for example

$$\begin{aligned} \mathsf{L}(\mathsf{N}, 1) &\rightsquigarrow \text{two copies of type } \mathsf{L}(\mathsf{N}) \\ \mathsf{L}(\mathsf{N} \otimes \mathsf{N}, 3) &\rightsquigarrow \text{two copies of type } \mathsf{L}(\mathsf{N} \otimes \mathsf{N}, 1) \\ \mathsf{L}(\mathsf{N} + (\mathsf{N}, 2), 9) &\rightsquigarrow \text{two copies of type } \mathsf{L}(\mathsf{N} + (\mathsf{N}, 2), 3) \end{aligned}$$

The 1-level Contraction for trees is similar with each node required to contain enough resources for two minor copies of itself. One can even imagine Contraction-rules of higher levels, e.g. allowing multiple use of variables of types like list of lists or tree of lists of trees, but note that already the first-level Contraction-rule destroys the validity of Theorem 2.4.20 regarding the feasibility of $(\!|P|\!)$ (while zero-level Contraction does not affect feasibility). The use of Contraction-rule of a level other than zero is even questionable as the next example shows.

Example 3.1.2 (Quick-Sort)**.** Take a look at the Quick-Sort implementation presented in [B&W88, p.154] translated into LF-style code:

$$\begin{aligned}
&\texttt{qsort} : (\mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N})\\
&\texttt{filter} : (\mathsf{L}(\mathsf{N}), \mathsf{N}+\mathsf{N}) \longrightarrow \mathsf{L}(\mathsf{N})
\end{aligned}$$

$$\begin{aligned}
\texttt{qsort}(l) = \mathsf{match}\ l\ \mathsf{with}&\\
&\mid \mathsf{nil} \Rightarrow \mathsf{nil}\\
&\mid \mathsf{cons}(h,t) \Rightarrow \quad \texttt{qsort}\big(\texttt{filter}(t, \mathsf{inl}(h))\big)\\
&\qquad\qquad\qquad +\!\!+\ \mathsf{cons}(h, \mathsf{nil})\\
&\qquad\qquad\qquad +\!\!+\ \texttt{qsort}\big(\texttt{filter}(t, \mathsf{inr}(h))\big)
\end{aligned}$$

$$\begin{aligned}
\texttt{filter}(l,k) = \mathsf{match}\ k\ \mathsf{with}&\\
&\mid \mathsf{inl}(x) \Rightarrow \mathsf{match}\ l\ \mathsf{with}\\
&\qquad \mid \mathsf{nil} \Rightarrow \mathsf{nil}\\
&\qquad \mid \mathsf{cons}(h,t) \Rightarrow \mathsf{if}\ h < x\ \mathsf{then}\ \mathsf{cons}(h, \texttt{filter}(t, \mathsf{inl}(x)))\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \texttt{filter}(t, \mathsf{inl}(x))\\
&\mid \mathsf{inr}(x) \Rightarrow \mathsf{match}\ l\ \mathsf{with}\\
&\qquad \mid \mathsf{nil} \Rightarrow \mathsf{nil}\\
&\qquad \mid \mathsf{cons}(h,t) \Rightarrow \mathsf{if}\ h < x\ \mathsf{then}\ \texttt{filter}(t, \mathsf{inr}(x))\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{cons}(h, \texttt{filter}(t, \mathsf{inr}(x)))
\end{aligned}$$

where $+\!\!+ : (\mathsf{L}(A), \mathsf{L}(A)) \longrightarrow \mathsf{L}(A)$ shall represent an infix operation of list concatenation, which could be added to LF and LFPL as a primitive operation without any problems. We introduced $+\!\!+$ here to stay as close as possible to the notation of `quicksort` in [B&W88] and also to keep the focus on the important parts of the example.

A plain translation of the code into LF, LFPL or $\widehat{\mathsf{LFPL}}$ is not possible, as the example is not linearly typable. The list t is passed twice to subsequent calls to the function `filter`, each of which processing the whole list. As `filter` constructs a new list, the argument is thus destroyed; otherwise `filter` would lack the resources for constructing its result list. In a non-linear language this problem simply does not arise, as arguments are copied anyway and in addition the allocation of new space resources is generally permitted.

Now even the 1-level Contraction rule would not be of help here: In each recursive step we would need another copy of the remaining list fragments. But note that copying

the list lowers the numbers of resources contained in each step, hence statically limiting the numbers of copies of type $\mathsf{L}(A,n)$ that can be obtained. However the numbers of copies needed in `quicksort` depends dynamically on length of the input list. Hence the 1-level Contraction rule is of no use in recursively defined functions, except for a terminating branch.

Of course, the algorithm quicksort can be easily implemented in $\widehat{\mathsf{LFPL}}$ – simply by replacing the use of `filter` (with mutually exclusive conditions) by using a function like $\texttt{split\_at} : \left(\mathsf{N}, \mathsf{L}(\mathsf{N},0), 0\right) \longrightarrow \mathsf{L}(\mathsf{N},0) \otimes \mathsf{L}(\mathsf{N},0)$ which splits the input list into two list by the given pivot – but it still requires a change to a more economic programming style, e.g. using `split_at` instead of `filter`:

Example 3.1.3 (Quick-Sort – $\widehat{\mathsf{LFPL}}$)**.**

$$\begin{aligned}
\texttt{qsort} &: (\mathsf{L}(\mathsf{N},0), 0) \longrightarrow \mathsf{L}(\mathsf{N},0) \\
\texttt{split_at} &: (\mathsf{N}, \mathsf{L}(\mathsf{N},0), 0) \longrightarrow \mathsf{L}(\mathsf{N},0) \otimes \mathsf{L}(\mathsf{N},0)
\end{aligned}$$

$$\begin{aligned}
&\texttt{qsort}(l) = \mathsf{match}\ l\ \mathsf{with} \\
&\qquad \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
&\qquad \mid \mathsf{cons}(h,t) \Rightarrow \\
&\qquad\quad \mathsf{match}\ \texttt{split_at}(h,t)\ \mathsf{with}\ u \otimes l \Rightarrow \\
&\qquad\qquad \texttt{qsort}(u) +\!\!+ \mathsf{cons}(h, \mathsf{nil}) +\!\!+ \texttt{qsort}(l)
\end{aligned}$$

$$\begin{aligned}
&\texttt{split_at}(p,l) = \mathsf{match}\ l\ \mathsf{with} \\
&\qquad \mid \mathsf{nil} \Rightarrow \mathsf{nil} \otimes \mathsf{nil} \\
&\qquad \mid \mathsf{cons}(h,t) \Rightarrow \\
&\qquad\quad \mathsf{match}\ \texttt{split_at}(p,t)\ \mathsf{with}\ u \otimes l \Rightarrow \\
&\qquad\qquad \mathsf{if}\ h \leq p\ \mathsf{then}\ \mathsf{cons}(h,u) \otimes l \\
&\qquad\qquad\qquad\quad \mathsf{else}\ \ u \otimes \mathsf{cons}(h,l)
\end{aligned}$$

Please note that the plain program code given above would also serve as another example being successfully handled by the methods of Section 1, when turning it into an LF program by changing the signature to

$$\begin{aligned}
\texttt{qsort} &: (\mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N}) \\
\texttt{split_at} &: (\mathsf{N}, \mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N}) \otimes \mathsf{L}(\mathsf{N})
\end{aligned}$$

and defining $(\!|e_1 +\!\!+ e_2|\!) = (\!|e_1|\!) + (\!|e_2|\!)$, as list-concatenation does not allocate any additional heap-space.

Before continuing with the next open question we should also mention that in the special case of a conditional operation, Martin Hofmann showed in [Hof00, Section 6 – "Additive rule for conditionals"] that it is in principle possible to allow variables of any type to be shared between guard and branches of a conditional, though with expensive cost. However, conditional operations are the most likely were a violation of a strict linear typing probably occurs, so combined with the contraction rules the comfort achieved might be worth considering.

3.2 Polymorphism with respect to $\Diamond$

In the previous example of `quicksort` another problem showed up: Even if the 1-level Contraction rule would help in principle, we would run into type problems with the recursive calls of the function `quicksort` to itself.

In general it might be possible that for a certain LF program P the integer linear program $(\!|P|\!)$ is only solvable if we allow multiple types for one and the same LF-function f, distinguished only by a different number of resources in its input.

Example 3.2.1. Recall the function `twice` mentioned in Example 1.9, which simply doubles each element of a list of natural numbers, provided that the 0-level Contraction rule is present as usual in our examples:

$$\begin{aligned}\texttt{twice} &: (\mathsf{L}(\mathsf{N},1),0) \to \mathsf{L}(\mathsf{N})\\ \texttt{twice}(l) &= \mathsf{match}\ l\ \mathsf{with}\ |\mathsf{nil} \Rightarrow \mathsf{nil}\\ &\qquad\quad |\mathsf{cons}(h,t) \Rightarrow \mathsf{cons}\big(h, \mathsf{cons}(h, \texttt{twice}(t))\big)\end{aligned}$$

Now for a function call like $\texttt{twice}\big(\texttt{twice}(l)\big)$ we would need to define the function `twice` twice, just with different $\widehat{\mathsf{LFPL}}$-signatures, but with the same defining body

$$\begin{aligned}\texttt{twice}_1 &: (\mathsf{L}(N,1),0) \to \mathsf{L}(N)\\ \texttt{twice}_2 &: (\mathsf{L}(N,3),0) \to \mathsf{L}(N,1)\end{aligned}$$

Note that this solution is unpleasing, as we want to implement in LF rather than directly in $\widehat{\mathsf{LFPL}}$, and hence cannot distinguish these functions. One could also imagine that there exists an example where it is impossible to use distinct function symbols, due to recursion and conditionals. So allowing the inference of types which are polymorphic in the number of resources could be very useful, resulting in the above example in the type

$$\texttt{twice} : (\mathsf{L}(N,2n+1),0) \to \mathsf{L}(N,n)$$

In this context we also conjecture that result values may always be resource free, i.e. that a type like $\mathsf{L}(N, n+1)$ will never occur on the right-hand side of first-order type. Thus resources in results are possibly only required if the result itself is used as an input for further computations. It might be interesting to investigate this further.

In his work [Hof00] Martin Hofmann already showed how general polymorphism and high-order functions could be included in LFPL, though we leave the integration of these features into $\widehat{\mathsf{LFPL}}$ – and the specialities arising like just described – open for future research.

3.3 Labelled Leaf Trees

We should also mention shortly the reason for switching to unlabelled leaf trees in our version of LFPL contrary to the labelled leaf tree type contained in [Hof00] and show how to avoid the problem:

Trees with labelled leafs cannot be allowed in $\widehat{\mathsf{LFPL}}$ as then we must change the Definition 2.1.1 of $\langle\cdot\rangle$:

$$\langle \mathsf{T}(A) \rangle = \langle A \rangle \neq 0$$

as any element of the tree type $\mathsf{T}(A)$ then contains at least one element of type A. Hence we must prohibit trees as return types of functions or risk losing the faithfulness of the signature, which was essential for our approach of the general problem.

Of course we could have introduced two-sorted trees $\mathsf{T}(R, P)$ in the type grammar of $\widehat{\mathsf{LFPL}}$, where the leaves are labelled by pure zero-order types and only the nodes are allowed to contain rich zero-order types, which avoids the problem depicted above. An element of $\mathsf{T}(A, B)$ could then possibly be a leaf with an element of type B, but since B must be a pure type satisfying $\langle B \rangle = 0$, the minimal content of the tree then is $\langle \mathsf{T}(A, B) \rangle = \min\big(\langle A \rangle, \langle B \rangle\big) = \min\big(\langle A \rangle, 0\big) = 0$ as required.

The two-sorted tree simply seemed unappealing to introduce, as its notation is farther away from the tree type presented in [Hof00] than the unlabelled leaf tree. Although it would have been nice for Example 2.4.23 constructing the Huffman tree.

3.4 Computing the heap-space usage from $\widehat{\mathsf{LFPL}}$ signatures

Once we know a valid $\widehat{\mathsf{LFPL}}$ (or $\widehat{\mathsf{LFPL}}_R$) signature for an LF program P, we do not exactly know its heap-space usage – we merely know how to compute it! Let us again consider the simple example of the list-node doubling function `twice` from

section 3.2: From the signature `twice` : $(\mathsf{L}(N,1),0) \to \mathsf{L}(N)$ we may deduce at a glance that computing `twice` requires an additional amount of heap-space equal to the size of the input list, since each list-node (occupying one $\Diamond$) must hold one spare resources (again the size of a $\Diamond$). So once we know the specific input and hence the length of the list, we know the heap-space used up by a computation of `twice`.

Since all initial data structures for P must be provided externally, any useful implementation of $\widehat{\mathsf{LFPL}}$ must therefore provide an automatism that, given a concrete data structure to be applied, computes the amount of heap-space additionally needed to apply P to this data (and allocates it). Computing the overall additional heap-space usage for e.g. a list of trees of lists, once given the length of each list, the size of each tree and the LFPL signature of the function the data is to be applied to, is certainly not a difficult task, yet it remains to be done.

3.5 Conclusion

With these questions just partly discussed, we close our examination of the problem here. From a practical viewpoint the achievements already made seem quite promising: The presented language $\widehat{\mathsf{LFPL}}$ reduces the resource handling of LFPL to its very essence without apparently losing expressive power as the provided examples show, hence we are left with a mathematically well manageable system in respect to the normalised resource management.

The results that constructing and solving the occurring integer linear programs are feasible and complete (in the sense that we will obtain a $\widehat{\mathsf{LFPL}}_R$ translation whenever one exists) furthermore promises a useful implementation. One could imagine an interactive environment for programming in a LF-like language that – combined with a solver for linear programs – already automatically monitors the consumption on the heap-space while a program is being implemented. Then whenever solving the ILP fails, examining its structure and the violated inequalities might help to easily identify the problem and the source of a memory leak.

Hence constructing linear functional programs that guarantee a limit on their heap-space usage becomes even more conveniently realizable at little more effort. Of course there are even more issues to be settled to add usability, e.g. neglecting the limitation to first-order functions. In this specific example, Eike Ritter [Rit00] showed how to include high-order functions in LFPL, yet it is not entirely clear whether or not such techniques can be adopted to our presented approach without failing or losing feasibility.

Appendix

A.1 Common notions

Stack-/Heap-space and Pointers: In classical programming languages like C, the memory usage is distinguished into two kinds:

- The space required to hold the values of a subroutine's local variables is called the *stack*. Whenever a subroutine is started, the stack grows to hold the local variables of fixed size (booleans, integers, pointers,...). When the subroutine is finished that portion becomes free again. Due to the nature of nested subroutine calls, the portion that has been allocated at last is the portion to be released first, which makes it very easy to organise the stack usage efficiently.
- All data structures of varying size such as lists or trees are placed in a part of the memory called *heap*. Those parts are usually addressed by the use of *pointers* which are data structures of small determined size just storing the memory address where such a certain portion of heap space starts (e.g. the head of a list). Since both the size and the life-time of heap-space structures are unknown (e.g. lists may be passed on to subroutines which may change their length, etc.), organising the heap efficiently is a more difficult task. Note furthermore that pointers have to be used with great care to avoid aliasing effects.

Garbage Collection: During a program's execution both stack and heap may grow. Since memory resources are limited, as soon as there is no room for further growth a program may fail due to the lack of free memory. Therefore it is important to efficiently organise the heap, so that data structures are immediately removed when they are not required any longer. This is not a trivial task and in some languages, like C, the task of deleting obsolete data is therefore left entirely to the programmer, which usually requires noticeable effort keeping track of the lifetime of each data object.

On the other hand there are languages, like Java, which try to ease the programmer's burden and provide an automatic management of the heap-space. An algorithm usually called *Garbage Collection* is periodically triggered and tries to determine which data structures have become obsolete, e.g. by tracing the present pointers, so that the space occupied by these may be reused.

There are of course some problems to be faced to construct a sensible Garbage Collection: Deleting data still in the program's use is certainly hazardous, and

a Garbage Collection cannot safely delete a data structure from the heap if there are still pointers left pointing to that structure, since it cannot determine (without a program analysis) whether or not a program will use these pointers. Note that pointers are usually also stored within the heap-space, e.g. each list-node contains a pointer to the next list-node, while only the head of the list may be stored in the stack or again within another heap-space data structure like a list of lists. Even if a direct pointer does not exist at all, the data might still be needed, e.g. it is not unusual in C to increment, decrement or even add and subtract pointers to address arrays, although arrays can be considered as a block and any pointer pointing somewhere inside this block can be regarded as a direct pointer towards the array. Another problem arises in the form of circular structures, e.g. a circular list, which may not be used anymore, but cannot be safely deleted either since it contains pointers to itself.

So in general any safe Garbage Collection will not be able to reclaim the maximal space that *is* (semantically) garbage. Programs heavily relying on a Garbage Collection, like interpreters for functional languages, therefore eventually run out of memory space during long computations.

See also [T&T97, Introduction] for a fine explanation of the general problem.

Aliasing: Consider a function which should take a list as an argument and returns the tail of the list. In C we could simply implement this function by returning a pointer to the second list-node. If the program manipulates afterwards the original list, also the tail is manipulated, since both pointers refer to the same memory locations. Whenever such effects due to multiple pointers towards one memory location occur unwontedly, we call them *aliasing* side effects.

Signature:[17] A *signature*, usually denoted by the symbol Σ, is a map from a finite set of function identifiers to the set of types of a certain programming language like LFPL or LF. The type then usually denotes the input and the output of the function referred to by its identifier.

Typing context:[17] A *typing context*, usually denoted by the symbol Γ, maps a finite set of variable identifiers to the set of types of a programming language. When dealing with program fragments it is necessary to record the type of a variable present in that program fragments scope. This is usually done by extending the current context.

[17] This shall only give a rough general intuition of the notion when used in the context of a specific functional programming language like LFPL or LF. A language definition must contain the precise definition if that concept is to be used for that particular language.

A common notation for a context where x has type N and y has type $\mathsf{N} \otimes \mathsf{N}$ is $\{x : \mathsf{N}, y : \mathsf{N} \otimes \mathsf{N}\}$. We often omit the brackets. If Γ_1 and Γ_2 denote contexts of the same programming language having *disjoint domains*, we simply write Γ_1, Γ_2 to denote the context consisting of the disjoint union of Γ_1 and Γ_2.

Typing judgement: A *typing judgement* written as $\Gamma \vdash_\Sigma e : A$, means that the program term e has the type A under the signature Σ and typing context Γ.

For a specific programming language, usually a set of typing rules state which typing judgements are true. We use the well-known notation

$$\frac{\text{Typing judgments required}}{\text{Typing judgment derived}} \text{ (Side conditions)}$$

Linear typing: A *linear typing* means that a variable may be used at most once. Typing rules of a linearly typed language have the form

$$\frac{\Gamma_i \vdash_\Sigma e_i : A_i \quad (i = 1, \ldots, n)}{\Gamma_1, \ldots, \Gamma_n \vdash_\Sigma e} \qquad \text{compared to} \qquad \frac{\Gamma \vdash_\Sigma e_i : A_i \quad (i = 1, \ldots, n)}{\Gamma \vdash_\Sigma e}$$

where the e_i are sub-terms of e. Thus the context of e is split up for the sub-terms, whereas in the right example each sub-term might use all variables contained in the context of e. Note that even in linearly typed languages typing judgements resembling the right side might occur: For example a conditional operation might allow a variable to be shared among its branches, as only one branch will be executed.

A linear typing is, for an example, necessary in languages like LFPL, where a resource type $\Diamond$ exists which can only be used once. A non-linear typing would otherwise allow the copying of resources, clearly rendering the concept of resources useless.

For a sensible non-linear example please see Example 3.1.2 (Quick-Sort) and see how the algorithm may be reformulated in a linear manner in this specific case (Example 3.1.3).

Program: A *program*, denoted by P, consists of a signature Σ and a collection of terms e_f for each function symbol $f \in \mathrm{dom}(\Sigma)$, for each of which the typing judgement $\Gamma \vdash_\Sigma \epsilon_f : A$ can be derived with Γ and A according to $\Sigma(f)$.

For a specific language like LFPL this will be formulated more precisely as

$$\forall f \in \mathrm{dom}(\Sigma)\,. \quad \Sigma(f) = (A_1, \ldots, A_n) \longrightarrow B \quad \Longrightarrow \quad v_1 : A_1, \ldots, v_n : A_n \overset{\mathsf{LFPL}}{\vdash_\Sigma} e_f : B$$

where the v_i are the only free variables occurring in e_f.

Branching operation: By a *branching operation* we mean any program term whose evaluation a priori requires only the evaluation of a part of its sub-terms, e.g. the branching operations of LFPL are Conditional, Sum-, List- and Tree-Elimination.

A List-Elimination operation like $\mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{nil} \Rightarrow e_2\ |\mathsf{cons}(h,t) \Rightarrow e_3$ is a branching operation because it distinguishes whether the expression e_1 is an empty list or not. In the first case of e_1 being equal to the empty list nil the whole expression evaluates to e_2. Otherwise e_1 is a non-empty list, hence of the form $\mathsf{cons}(h,t)$, where h is the head of the list and t its tail. The whole expression then evaluates to e_3, which may contain h and t as free variables which then assume the appropriate values. The other elimination operations function similarly.

Shorthands for Lists: In the presented examples we use the shorthands $\mathsf{nil} = [\,]$ and $\mathsf{cons}\big(1, \mathsf{cons}(2, \mathsf{nil})\big) = 1 :: 2 :: [\,] = [1,2]$. Furthermore $+\!\!+$ denotes the concatenation of lists, e.g.: $[1,2,3] +\!\!+ [4,5,6] = [1,2,3,4,5,6]$.

A.2 The language LFPL

LFPL is a Linearly typed first-order Functional Programming Language, presented by Martin Hofmann in [Hof00]. The peculiarity of LFPL is the resource type $\Diamond$, which allows a translation of LFPL into malloc()-free C Code. Please see the Introduction of this work for more explanations about this feature or examine the provided examples at the end of this section.

We deal with a slight variant of LFPL here, compared to what is written in [Hof00]. The changes are:

- We add the unit type 1 with its single element denoted by $*$ as another base type.
- We do not mention the Contraction-rule until introduced in Section 3.1, though we implicitly use it throughout the presented examples. The lack of the Contraction-rule does not limit the expressive power of LFPL, the rule rather provides convenience.
- The recursive binary tree type $\mathsf{T}(\cdot)$ has unlabelled leafs only, hence constructing a tree node requires only one element of type $\Diamond$, since the subtrees of a node are reducible to pointers then and the one resource required is just needed to hold the tree-node itself. (see Section 3.3 on labelled-leaf trees).

Definition A.2.1 (LFPL)**.** The LFPL type grammar:

$$\begin{array}{ll} \text{zero-order types:} & A ::= \mathsf{1} \mid \mathsf{N} \mid \Diamond \mid \mathsf{L}(A) \mid \mathsf{T}(A) \mid A \otimes A \mid A + A \\ \text{first-order types:} & F ::= (A, \ldots, A) \to A \end{array}$$

The terms of LFPL are given by the following grammar:

$$\begin{array}{lll} e ::= & v & \text{Variable} \\ \mid & c & \text{Constant} \\ \mid & e_1 \star e_2 \qquad \big(\text{for } \star \in \{+, -, \times, \geq, \ldots\}\big) & \text{Standard Integer Infix} \\ \mid & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \text{Conditional} \\ \mid & e_1 \otimes e_2 & \text{Pairing} \\ \mid & \mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2 & \text{Pair-Elimination} \\ \mid & \mathsf{inl}(e) & \text{Left Injection} \\ \mid & \mathsf{inr}(e) & \text{Right Injection} \\ \mid & \mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{inl}(v) \Rightarrow e_2 & \text{Sum-Elimination} \\ & \quad\ \ \mathsf{inr}(v) \Rightarrow e_3 & \\ \mid & \mathsf{nil} & \text{Empty List} \\ \mid & \mathsf{cons}(e_d, e_h, e_t) & \text{List-Construction} \\ \mid & \mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_2 & \text{List-Elimination} \\ & \quad\ \ \mathsf{cons}(d, h, t) \Rightarrow e_3 & \\ \mid & \mathsf{leaf} & \text{Leaf} \\ \mid & \mathsf{node}(e_d, e_a, e_l, e_r) & \text{Tree-Node} \\ \mid & \mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{leaf} \Rightarrow e_2 & \text{Tree-Elimination} \\ & \quad\ \ \mathsf{node}(d, a, l, r) \Rightarrow e_3 & \\ \mid & f(e_1, \ldots, e_n) & \text{Function Application} \end{array}$$

In each of the following type rules, let Σ denote a LFPL signature mapping a finite set of function identifiers to LFPL first-order types, Γ be a LFPL typing context mapping a finite set of identifiers to LFPL zero-order types, $e, e_a, e_b, \ldots$ representing arbitrary LFPL terms according to the grammar stated above, and A, B, C denoting arbitrary LFPL zero-order types. We allow the following typing judgements for LFPL provided that the premises and side conditions are met.

Variable

$$\frac{}{\Gamma \vdash_\Sigma v : \Gamma(v)}\ (v \in \mathrm{dom}(\Gamma))$$

Constant I+II

$$\frac{}{\Gamma \vdash_\Sigma * : 1} \qquad \frac{}{\Gamma \vdash_\Sigma c : \mathsf{N}}\ (c \text{ is a integer constant})$$

Standard Integer Infix Operator

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : \mathsf{N} \qquad \Gamma_2 \vdash_\Sigma e_2 : \mathsf{N}}{\Gamma_1, \Gamma_2 \vdash_\Sigma e_1 \star e_2 : \mathsf{N}} \begin{pmatrix} \star \text{ is a integer} \\ \text{infix operator} \end{pmatrix}$$

Conditional

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : \mathsf{N} \qquad \Gamma_2 \vdash_\Sigma e_2 : A \qquad \Gamma_2 \vdash_\Sigma e_3 : A}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A}$$

Pairing

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : A_1 \qquad \Gamma_2 \vdash_\Sigma e_2 : A_2}{\Gamma_1, \Gamma_2 \vdash_\Sigma e_1 \otimes e_2 : A_1 \otimes A_2}$$

Pair-Elimination

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : A_1 \otimes A_2 \qquad \Gamma_2, v_1 : A_1, v_2 : A_2 \vdash_\Sigma e_2 : C}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2 : C}$$

Inl, Inr

$$\frac{\Gamma \vdash_\Sigma e : A}{\Gamma \vdash_\Sigma \mathsf{inl}(e) : A + B} \qquad \frac{\Gamma \vdash_\Sigma e : B}{\Gamma \vdash_\Sigma \mathsf{inr}(e) : A + B}$$

Sum-Elimination

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : A + B \qquad \Gamma_2, v : A \vdash_\Sigma e_2 : C \qquad \Gamma_2, v : B \vdash_\Sigma e_3 : C}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{inl}(v) \Rightarrow e_2\ |\mathsf{inr}(v) \Rightarrow e_3 : C}$$

Empty List

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{nil} : \mathsf{L}(A)}$$

List-Construction

$$\frac{\Gamma_d \vdash_\Sigma e_d : \Diamond \qquad \Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)}$$

List-Elimination

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : \mathsf{L}(A) \qquad \Gamma_2 \vdash_\Sigma e_2 : C \qquad \Gamma_2, d : \Diamond, h : A, t : \mathsf{L}(A) \vdash_\Sigma e_3 : C}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ \mid nil \Rightarrow e_2 \mid \mathsf{cons}(d, h, t) \Rightarrow e_3 : C}$$

Leaf

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{leaf} : \mathsf{T}(A)}$$

Tree-Node

$$\frac{\Gamma_d \vdash_\Sigma e_d : \Diamond \qquad \Gamma_a \vdash_\Sigma e_a : A \qquad \begin{array}{r}\Gamma_l \vdash_\Sigma e_l : \mathsf{T}(A) \\ \Gamma_r \vdash_\Sigma e_r : \mathsf{T}(A)\end{array}}{\Gamma_d, \Gamma_a, \Gamma_l, \Gamma_r \vdash_\Sigma \mathsf{node}(e_d, e_a, e_l, e_r) : \mathsf{T}(A)}$$

Tree-Elimination

$$\frac{\begin{array}{c}\Gamma_1 \vdash_\Sigma e_1 : \mathsf{T}(A) \qquad \Gamma_2 \vdash_\Sigma e_2 : C \\ \Gamma_2, d : \Diamond, a : A, l : \mathsf{T}(A), r : \mathsf{T}(A) \vdash_\Sigma e_3 : C\end{array}}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{leaf} \Rightarrow e_2 \mid \mathsf{node}(d, a, l, r) \Rightarrow e_3 : C}$$

Function Application

$$\frac{\Sigma(f) = (A_1, \ldots, A_p) \longrightarrow B \qquad \Gamma_i \vdash_\Sigma e_i : A_i \text{ for } i = 1, \ldots, p}{\Gamma_1, \ldots, \Gamma_p \vdash_\Sigma f(e_1, \ldots, e_p) : B}$$

Standard Set-Theoretic Semantics of LFPL

The standard set-theoretic semantics of LFPL is given in [Hof00, 3.2]. Although we do not make explicit use of the set-theoretic semantics of LFPL throughout this work, we restate them here for the used variant of LFPL and for your convenience. Note that the set-theoretic interpretation given ignores all resource related issues connected with type $\Diamond$; its purpose is merely to provide a formal functional denotation of LFPL programs.

Definition A.2.2 (Set-theoretic interpretation of LFPL)**.** The LFPL types are interpreted by the following sets:

$$\begin{aligned}
[\![\mathsf{1}]\!] &= \{0\} \\
[\![\mathsf{N}]\!] &= \mathbb{Z} \\
[\![\Diamond]\!] &= \{0\} \\
[\![\mathsf{L}(A)]\!] &= \text{finite lists over } [\![A]\!] \\
[\![\mathsf{T}(A)]\!] &= \text{finite binary trees with} \\
&\qquad [\![A]\!]\text{-labelled nodes and unlabelled leaves} \\
[\![A \otimes B]\!] &= [\![A]\!] \times [\![B]\!] \\
[\![A + B]\!] &= \big\{\mathrm{inl}(a) \;\big|\; a \in [\![A]\!]\big\} \cup \big\{\mathrm{inr}(b) \;\big|\; b \in [\![B]\!]\big\} \\
[\![A_1, \ldots, A_n \to B]\!] &= \text{partial functions from } [\![A_1]\!] \times \cdots \times [\![A_n]\!] \text{ to } [\![B]\!]
\end{aligned}$$

A function η is a *valuation of a context* Γ if $\eta(v) \in [\![\Gamma(v)]\!]$ holds for each $v \in \mathrm{dom}(\Gamma)$; a function ρ is a *valuation of a signature* Σ if $\rho(f) \in [\![\Sigma(f)]\!]$ holds for all $f \in \mathrm{dom}(\Sigma)$.

Given both a valuation η of the context Γ and a valuation ρ of the signature Σ, we can define the meaning $[\![e]\!]_{\eta,\rho} \in [\![A]\!] \cup \{\bot\}$ of an LFPL term e provided that $\Gamma \vdash_\Sigma e : A$ holds, otherwise $[\![e]\!]_{\eta,\rho} = \bot$. The mapping $[\![\cdot]\!]_{\eta,\rho}$ is then defined inductively on the composition of LFPL terms as follows:

$$
\begin{aligned}
[\![v]\!]_{\eta,\rho} &= \eta(v) \\
[\![c]\!]_{\eta,\rho} &= c \\
[\![e_1 \star e_2]\!]_{\eta,\rho} &= [\![e_1]\!]_{\eta,\rho} \star [\![e_2]\!]_{\eta,\rho} \\
[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]_{\eta,\rho} &= \begin{cases} [\![e_2]\!]_{\eta,\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = 0 \\ [\![e_3]\!]_{\eta,\rho} & \text{otherwise} \end{cases} \\
[\![e_1 \otimes e_2]\!]_{\eta,\rho} &= \big([\![e_1]\!]_{\eta,\rho}, [\![e_2]\!]_{\eta,\rho} \big) \\
[\![\mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2]\!]_{\eta,\rho} &= [\![e_2]\!]_{\eta\cup\{v_1\mapsto a, v_2\mapsto b\},\rho} \text{ for } [\![e_1]\!]_{\eta,\rho} = (a,b) \\
[\![\mathsf{inl}(e)]\!]_{\eta,\rho} &= \mathrm{inl}\big([\![e]\!]_{\eta,\rho}\big) \\
[\![\mathsf{inr}(e)]\!]_{\eta,\rho} &= \mathrm{inr}\big([\![e]\!]_{\eta,\rho}\big) \\
\Big[\!\Big[\begin{array}{r} \mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{inl}(v) \Rightarrow e_2 \\ \mid \mathsf{inr}(v) \Rightarrow e_3 \end{array} \Big]\!\Big]_{\eta,\rho} &= \begin{cases} [\![e_2]\!]_{\eta\cup\{v\mapsto a\},\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = \mathrm{inl}(a) \\ [\![e_3]\!]_{\eta\cup\{v\mapsto a\},\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = \mathrm{inr}(a) \end{cases} \\
[\![\mathsf{nil}]\!]_{\eta,\rho} &= [\,] \\
[\![\mathsf{cons}(e_d, e_h, e_t)]\!]_{\eta,\rho} &= [\![e_h]\!]_{\eta,\rho} :: [\![e_t]\!]_{\eta,\rho} \\
\Big[\!\Big[\begin{array}{l} \mathsf{match}\ e_1\ \mathsf{with} \\ \quad \mid \mathsf{nil} \Rightarrow e_2 \\ \quad \mid \mathsf{cons}(d,h,t) \Rightarrow e_3 \end{array} \Big]\!\Big]_{\eta,\rho} &= \begin{cases} [\![e_2]\!]_{\eta,\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = [\,] \\ [\![e_3]\!]_{\eta',\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = a :: l \\ & \quad \text{and } \eta' = \eta \cup \left\{ \begin{array}{l} d \mapsto 0 \\ h \mapsto a \\ t \mapsto l \end{array} \right\} \end{cases} \\
[\![\mathsf{leaf}]\!]_{\eta,\rho} &= \mathrm{leaf} \\
[\![\mathsf{node}(e_d, e_a, e_l, e_r)]\!]_{\eta,\rho} &= \mathrm{node}\big([\![e_a]\!]_{\eta,\rho}, [\![e_l]\!]_{\eta,\rho}, [\![e_r]\!]_{\eta,\rho}\big) \\
\Big[\!\Big[\begin{array}{l} \mathsf{match}\ e_1\ \mathsf{with} \\ \quad \mid \mathsf{leaf} \Rightarrow e_2 \\ \quad \mid \mathsf{node}(d,a,l,r) \Rightarrow e_3 \end{array} \Big]\!\Big]_{\eta,\rho} &= \begin{cases} [\![e_2]\!]_{\eta,\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = \mathrm{leaf} \\ [\![e_3]\!]_{\eta',\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = \mathrm{node}(u,v,w) \\ & \quad \text{and } \eta' = \eta \cup \left\{ \begin{array}{l} d \mapsto 0 \\ a \mapsto u \\ l \mapsto v \\ r \mapsto w \end{array} \right\} \end{cases} \\
[\![f(e_1, \ldots, e_n)]\!]_{\eta,\rho} &= \rho\big(f\big)\big([\![e_1]\!]_{\eta,\rho}, \ldots, [\![e_n]\!]_{\eta,\rho}\big)
\end{aligned}
$$

An LFPL program P with signature Σ, and for all $f \in \mathrm{dom}(\Sigma)$ the defining term of function f being e_f, is then interpreted as the least valuation ρ such that for all functions $f \in \mathrm{dom}(\Sigma)$ holds $\rho(f)(x_1, \ldots, x_n) = [\![e_f]\!]_{\eta,\rho}$ where $\eta(v_i) = x_i$.

Program Examples

For getting a good intuition how the programming language LFPL works, please consider the following short program examples. Note that the LF translation of each program is also given at the end of Appendix A.3.

Example A.2.3 (Reverse – LFPL)**.** The function `reverse` takes a list of natural numbers and reverses its order. The recursion is programmed in accumulator style via the function `rev_aux` which receives another list representing the partial result (the accumulator) as a second argument. (See Example A.3.3 for a detailed step by step example computation of `reverse`.)

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

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

Please note how the resource $(d : \Diamond)$ from splitting the head of the argument list is used to construct the resulting list: In the C translation given by Martin Hofmann [Hof00], each element of type $\Diamond$ corresponds to a small distinct portion of heap-space, just large enough to hold a single list- or tree-node. Splitting a list into head and tail returns an element of type $\Diamond$, which corresponds to the location where the head was stored on the heap (at run-time of the C translation). Since the linear typing rules deny further access to the whole list (l) after the splitting match operation, and allows only access to the remaining tail (t), we may reuse the heap location of the head now for another purpose. (It *may* also be used to rebuild the former list again, hence simulating a read-only access.) So in this case we see that the execution of the C translation of `reverse` does not consume any additional heap-space, as the output is constructed by the use of the heap-space locations freed by splitting the input. We say that the output is constructed in-place. Of course, these observations only apply with the specific C translation in mind.

Example A.2.4 (Conway's Sequence – LFPL)**.** The language LFPL allows functions with dynamical resource consumption depending on their input. As an example we

consider the computation of Conway's Sequence [Con87]:

$$[1], [1,1], [2,1], [1,2,1,1], [1,1,1,2,2,1], [3,1,2,2,1,1], [1,3,1,1,2,2,2,1], \ldots$$

Each successor can be viewed as the proverbial description of its predecessor: "The list $[2,1]$ consists of *one* 2 followed by *one* 1" $\approx [1,2,1,1]$

Below we will define the function `verbal`, which computes the successor of a given list, e.g. $\texttt{verbal}\big([2,1]\big) = [1,2,1,1]$. Note that the result in this case is twice the size of the input, hence we cannot hope to compute `verbal` without allocating new resources. (In fact, the growth is exponential: $C \cdot \lambda^n$ for $\lambda \approx 1.303577269$, see [Var91] for details.) However, as pointed out in the introduction, LFPL does not allow the allocation of new memory resources! Let us see first how the problem arises:

Warning: The following lines are an example of *incorrect* LFPL code. It demonstrates the difficulties arising when translating LF naively into LFPL, as the shown code is almost identical to the *valid* LF code of the same function, given in Example A.3.4.

$$\begin{array}{l}
\texttt{verbal}' : (\mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N}) \\
\texttt{vbaux}' : (\mathsf{N}, \mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N}) \\
\\
\texttt{verbal}'(l) = \texttt{vbaux}'(1, l) \\
\texttt{vbaux}'(n, l) = \ \mathsf{match}\ l\ \mathsf{with} \\
\qquad\qquad \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
\qquad\qquad \mid \mathsf{cons}(d_1, c_1, t_1) \Rightarrow \\
\qquad\qquad\quad \mathsf{match}\ t_1\ \mathsf{with} \\
\qquad\qquad\quad \mid \mathsf{nil} \Rightarrow \mathsf{cons}\big(d_1, n, \mathsf{cons}(d_1, c_1, \mathsf{nil})\big) \\
\qquad\qquad\quad \mid \mathsf{cons}(d_2, c_2, t_2) \Rightarrow \\
\qquad\qquad\qquad \mathsf{if}\ c_1 = c_2 \\
\qquad\qquad\qquad \mathsf{then}\ \texttt{vbaux}'\big(n+1, \mathsf{cons}(d_1, h_2 \otimes d_2, t_2)\big) \\
\qquad\qquad\qquad \mathsf{else}\ \mathsf{cons}\big(d_1, n, \mathsf{cons}\big(d_1, h_1, \texttt{vbaux}'\big(1, \mathsf{cons}(d_2, h_2 \otimes d_2, t_2)\big)\big)\big)
\end{array}$$

This is incorrect code because it is not linearly typable, hence not a well-typed program according to Definition A.2.1: In the 5^{th} line of the definition of function `vbaux` the variable d_1 is used twice, once as the first sub-term of the cons function and once again within its third sub-term. The type rule for List-Construction requires that the contexts needed to type both sub-terms must be disjoint, which is violated here. (As explained in Appendix A.1, we write Γ_1, Γ_2 for the union of Γ_1 and Γ_2 only if Γ_1 and Γ_2 are disjoint.) In addition, in the last line of the code the variables d_1 and d_2 are both used twice, violating linearity again.

We promised that LFPL allows the dynamical allocation memory resources, so let us see now how `verbal` can be implemented in a well-typed LFPL program:

$$
\begin{array}{l}
\texttt{verbal} : (\mathsf{L}(\mathsf{N} \otimes \Diamond)) \longrightarrow \mathsf{L}(\mathsf{N}) \\
\texttt{vbaux} : (\mathsf{N}, \mathsf{L}(\mathsf{N} \otimes \Diamond)) \longrightarrow \mathsf{L}(\mathsf{N}) \\
\\
\texttt{verbal}(l) = \texttt{vbaux}(1, l) \\
\texttt{vbaux}(n, l) = \\
\quad \mathsf{match}\ l\ \mathsf{with} \\
\quad \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
\quad \mid \mathsf{cons}(d_{11}, h_1, t_1) \Rightarrow \mathsf{match}\ h_1\ \mathsf{with}\ d_{12} \otimes c_1 \Rightarrow \\
\qquad \mathsf{match}\ t_1\ \mathsf{with} \\
\qquad \mid \mathsf{nil} \Rightarrow \mathsf{cons}\big(d_{12}, n, \mathsf{cons}(d_{11}, c_1, \mathsf{nil})\big) \\
\qquad \mid \mathsf{cons}(d_{21}, h_2, t_2) \Rightarrow \mathsf{match}\ h_2\ \mathsf{with}\ d_{22} \otimes c_2 \Rightarrow \\
\qquad\quad \mathsf{if}\ c_1 = c_2 \\
\qquad\quad \mathsf{then}\ \texttt{vbaux}\big(n+1, \mathsf{cons}(d_{21}, h_2 \otimes d_{22}, t_2)\big) \\
\qquad\quad \mathsf{else}\ \ \mathsf{cons}\big(d_{12}, n, \mathsf{cons}\big(d_{11}, h_1, \texttt{vbaux}\big(1, \mathsf{cons}(d_{21}, h_2 \otimes d_{22}, t_2)\big)\big)\big)
\end{array}
$$

So each node of the input list is required to hold an additional spare resource, which thus allows us to construct an output list of at most twice the size of the input. The function `verbal` thus demands the *preallocation* of the additional resources needed, and then uses in-place updates of the input only. Therefore a malloc()-free C compilation can be provided. The requirement for preallocation of memory can always be easily spotted within a functions LFPL signature, by the presence or absence of the type $\Diamond$. Unlike in the shown example, some functions may require the preallocation of a fixed amount of memory, e.g. a call to a function of type $(\mathsf{N}, \mathsf{L}(\mathsf{N}), \Diamond) \to \mathsf{L}(\mathsf{N})$ would always consume two resources. The function `ins′` from Example 1.8 is of that type.

Throughout this work, we will use a very simple function called `twice` instead of `verbal` to demonstrate the dynamical allocation of memory depending on the input (Example 1.9 and Example 3.2.1). Rather than computing a specific series, `twice` merely doubles each list-node, but its signature is equal to `verbal`.

Alas, there is another problem we did not mention so far with this technique: A call like $\texttt{verbal}\big(\texttt{verbal}([1,1])\big)$ is invalid in LFPL:

a) The first reason being that $[1,1]$ should be a constant of type $\mathsf{L}(\mathsf{N} \otimes \Diamond)$ different from nil, which we simply cannot define within LFPL. We *cannot* even define

constants of type $\mathsf{L}(\mathsf{N})$, apart from nil:

$$\begin{aligned} \mathtt{naturals} &: (\mathsf{N}) \longrightarrow \mathsf{L}(\mathsf{N}) \\ \mathtt{naturals}(n) &= \mathsf{if}\ (n = 0)\ \mathsf{then\ nil\ else\ cons}\big(\_, n, \mathtt{natural}(n-1)\big) \end{aligned}$$

Apart from violating linearity, which is non-lethal in the case of N as we will see in Section 3.1, we cannot provide an element of type $\Diamond$ to perform the cons operation, as there are no constants of type $\Diamond$ available in LFPL. Hence any data structures must be provided externally to an LFPL program, by a mechanism which cares for the controlled allocation of memory. We then guarantee that executing a LFPL program on these data structures will not require any additional memory resources.

b) In the above example, one may argue that we could change the signature to $\mathtt{naturals} : (\mathsf{N}, \Diamond) \longrightarrow \mathsf{L}(\mathsf{N})$, which would provide the required resource. Now in this case the program would not be typable again, as the recursive call would then also require an element of type $\Diamond$.

In a similar manner, we cannot use `verbal` twice on the same data structure, as the first call changes its type from $\mathsf{L}(\mathsf{N} \otimes \Diamond)$ to $\mathsf{L}(\mathsf{N})$. Hence, in LFPL we cannot formulate a program that uses `verbal` recursively, like

$$\begin{aligned} \mathtt{iter\_map}_{\mathtt{verbal}}(l) = \mathsf{match}\ l\ \mathsf{with} \\ \quad &| \mathsf{nil} \Rightarrow \mathsf{nil} \\ \quad &| \mathsf{cons}(d, h, t) \Rightarrow \mathsf{cons}\big(d, h, \mathtt{verbal}(\mathtt{iter\_map}_{\mathtt{verbal}}(t))\big) \end{aligned}$$

However for $\mathtt{f} : (\mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N})$ we would obtain

$$\mathtt{iter\_map}_{\mathtt{f}}([a, b, c]) = a :: \mathtt{f}\big(b :: \mathtt{f}\big(c :: \mathtt{f}([\,])\big)\big)$$

as expected. We will discuss such recursion related type problems more closely in Section 3.2.

A.3 The language LF

LF is a linearly typed first-order functional programming language, similar to LFPL except for the resource type $\Diamond$, which is not included in LF. Therefore we cannot deduce much about the resource consumption of LF programs, contrary to LFPL programs. Please consider the provided examples at the end of this section to see what we receive in trade for giving up the type $\Diamond$.

Definition A.3.1 (LF)**.** The LF type grammar:

zero-order types:	$A ::= \mathsf{1} \mid \mathsf{N} \mid \mathsf{L}(A) \mid \mathsf{T}(A) \mid A \otimes A \mid A + A$
first-order types:	$F ::= (A, \ldots, A) \to A$

The terms of LF are given by the following grammar:

$e ::=$	v	Variable
	$\mid c$	Constant
	$\mid e_1 \star e_2 \qquad \big(\text{for } \star \in \{+, -, \times, \geq, \ldots\}\big)$	Standard Integer Infix
	$\mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$	Conditional
	$\mid e_1 \otimes e_2$	Pairing
	$\mid \mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2$	Pair-Elimination
	$\mid \mathsf{inl}(e)$	Left Injection
	$\mid \mathsf{inr}(e)$	Right Injection
	$\mid \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{inl}(v) \Rightarrow e_2$	Sum-Elimination
	$\qquad\qquad\qquad\quad \mid \mathsf{inr}(v) \Rightarrow e_3$	
	$\mid \mathsf{nil}$	Empty List
	$\mid \mathsf{cons}(e_1, e_2)$	List-Construction
	$\mid \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_2$	List-Elimination
	$\qquad\qquad\qquad\quad \mid \mathsf{cons}(h, t) \Rightarrow e_3$	
	$\mid \mathsf{leaf}$	Leaf
	$\mid \mathsf{node}(e_a, e_l, e_r)$	Tree-Node
	$\mid \mathsf{match}\ e_1\ \mathsf{with}\ \mid \mathsf{leaf} \Rightarrow e_2$	Tree-Elimination
	$\qquad\qquad\qquad\quad \mid \mathsf{node}(a, l, r) \Rightarrow e_3$	
	$\mid f(e_1, \ldots, e_n)$	Function Application

In each of the following type rules, let Σ denote a LF signature mapping a finite set of function identifiers to LF first-order types, Γ be a LF typing context mapping a finite set of identifiers to LF zero-order types, $e, e_a, e_b, \ldots$ representing arbitrary LF terms according to the given grammar, and A, B, C denoting arbitrary LF zero-order types. Then we allow the following typing judgements for LF provided that the premises and side conditions of each rule are met.

Variable

$$\frac{}{\Gamma \vdash_\Sigma v : \Gamma(v)}\ (v \in \mathrm{dom}(\Gamma))$$

Constant I+II

$$\frac{}{\Gamma \vdash_\Sigma * : 1} \qquad \frac{}{\Gamma \vdash_\Sigma c : \mathsf{N}}\ (c \text{ is a integer constant})$$

Standard Integer Infix Operator

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : \mathsf{N} \qquad \Gamma_2 \vdash_\Sigma e_2 : \mathsf{N}}{\Gamma_1, \Gamma_2 \vdash_\Sigma e_1 \star e_2 : \mathsf{N}} \begin{pmatrix} \star \text{ is a integer} \\ \text{infix operator} \end{pmatrix}$$

Conditional

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : \mathsf{N} \qquad \Gamma_2 \vdash_\Sigma e_2 : A \qquad \Gamma_2 \vdash_\Sigma e_3 : A}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A}$$

Pairing

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : A_1 \qquad \Gamma_2 \vdash_\Sigma e_2 : A_2}{\Gamma_1, \Gamma_2 \vdash_\Sigma e_1 \otimes e_2 : A_1 \otimes A_2}$$

Pair-Elimination

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : A_1 \otimes A_2 \qquad \Gamma_2, v_1 : A_1, v_2 : A_2 \vdash_\Sigma e_2 : C}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2 : C}$$

Inl, Inr

$$\frac{\Gamma \vdash_\Sigma e : A}{\Gamma \vdash_\Sigma \mathsf{inl}(e) : A + B} \qquad \frac{\Gamma \vdash_\Sigma e : B}{\Gamma \vdash_\Sigma \mathsf{inr}(e) : A + B}$$

Sum-Elimination

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : A + B \qquad \Gamma_2, v : A \vdash_\Sigma e_2 : C \qquad \Gamma_2, v : B \vdash_\Sigma e_3 : C}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{inl}(v) \Rightarrow e_2\ |\mathsf{inr}(v) \Rightarrow e_3 : C}$$

Empty List

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{nil} : \mathsf{L}(A)}$$

List-Construction

$$\frac{\Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A)}{\Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_h, e_t) : \mathsf{L}(A)}$$

List-Elimination

$$\frac{\Gamma_1 \vdash_\Sigma e_1 : \mathsf{L}(A) \qquad \Gamma_2 \vdash_\Sigma e_2 : C \qquad \Gamma_2, h : A, t : \mathsf{L}(A) \vdash_\Sigma e_3 : C}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ |nil \Rightarrow e_2\ |\mathsf{cons}(h,t) \Rightarrow e_3 : C}$$

Leaf

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{leaf} : \mathsf{T}(A)}$$

Tree-Node

$$\frac{\Gamma_a \vdash_\Sigma e_a : A \qquad \begin{array}{c} \Gamma_l \vdash_\Sigma e_l : \mathsf{T}(A) \\ \Gamma_r \vdash_\Sigma e_r : \mathsf{T}(A) \end{array}}{\Gamma_a, \Gamma_l, \Gamma_r \vdash_\Sigma \mathsf{node}(e_a, e_l, e_r) : \mathsf{T}(A)}$$

Tree-Elimination

$$\frac{\begin{array}{c} \Gamma_1 \vdash_\Sigma e_1 : \mathsf{T}(A) \qquad \Gamma_2 \vdash_\Sigma e_2 : C \\ \Gamma_2, a : A, l : \mathsf{T}(A), r : \mathsf{T}(A) \vdash_\Sigma e_3 : C \end{array}}{\Gamma_1, \Gamma_2 \vdash_\Sigma \mathsf{match}\ e_1\ \mathsf{with}\ |\mathsf{leaf} \Rightarrow e_2\ |\mathsf{node}(a,l,r) \Rightarrow e_3 : C}$$

Function Application

$$\frac{\Sigma(f) = (A_1, \ldots, A_p) \longrightarrow B \qquad \Gamma_i \vdash_\Sigma e_i : A_i \text{ for } i = 1, \ldots, p}{\Gamma_1, \ldots, \Gamma_p \vdash_\Sigma f(e_1, \ldots, e_p) : B}$$

Standard Set-Theoretic Semantics of LF

Since the given set-theoretic semantics of LFPL takes no account for the heap-space usage and hence rather ignores the type $\Diamond$, the set-theoretic semantics of LF are almost equal to those presented in Definition A.2.2. Nevertheless we state the set-theoretic semantics of LF here, especially since they are an *identical* part of the set-theoretic semantics of $\widehat{\mathsf{LFPL}}$.

Definition A.3.2 (Set-theoretic interpretation of LF)**.** The LF types are interpreted by the following sets:

$$\begin{aligned}
[\![\mathsf{1}]\!] &= \{0\} \\
[\![\mathsf{N}]\!] &= \mathbb{Z} \\
[\![\mathsf{L}(A)]\!] &= \text{finite lists over } [\![A]\!] \\
[\![\mathsf{T}(A)]\!] &= \text{finite binary trees with} \\
&\qquad [\![A]\!]\text{-labelled nodes and unlabelled leaves} \\
[\![A \otimes B]\!] &= [\![A]\!] \times [\![B]\!] \\
[\![A + B]\!] &= \big\{\mathrm{inl}(a) \;\big|\; a \in [\![A]\!]\big\} \cup \big\{\mathrm{inr}(b) \;\big|\; b \in [\![B]\!]\big\} \\
[\![A_1, \ldots, A_n \to B]\!] &= \text{partial functions from } [\![A_1]\!] \times \cdots \times [\![A_n]\!] \text{ to } [\![B]\!]
\end{aligned}$$

A function η is a *valuation of a context* Γ if $\eta(v) \in [\![\Gamma(v)]\!]$ holds for each $v \in \mathrm{dom}(\Gamma)$; a function ρ is a *valuation of a signature* Σ if $\rho(f) \in [\![\Sigma(f)]\!]$ holds for all $f \in \mathrm{dom}(\Sigma)$.

Given both a valuation η of the context Γ and a valuation ρ of the signature Σ, we can define the meaning $[\![e]\!]_{\eta,\rho} \in [\![A]\!] \cup \{\bot\}$ of an LF term e provided that $\Gamma \vdash_\Sigma e : A$ holds, otherwise $[\![e]\!]_{\eta,\rho} = \bot$. The mapping $[\![\cdot]\!]_{\eta,\rho}$ is then defined inductively on the composition of LF terms as follows:

$$
\begin{aligned}
[\![v]\!]_{\eta,\rho} &= \eta(v) \\
[\![c]\!]_{\eta,\rho} &= c \\
[\![e_1 \star e_2]\!]_{\eta,\rho} &= [\![e_1]\!]_{\eta,\rho} \star [\![e_2]\!]_{\eta,\rho} \\
[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]_{\eta,\rho} &= \begin{cases} [\![e_2]\!]_{\eta,\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = 0 \\ [\![e_3]\!]_{\eta,\rho} & \text{otherwise} \end{cases} \\
[\![e_1 \otimes e_2]\!]_{\eta,\rho} &= \big([\![e_1]\!]_{\eta,\rho}, [\![e_2]\!]_{\eta,\rho} \big) \\
[\![\mathsf{match}\ e_1\ \mathsf{with}\ v_1 \otimes v_2 \Rightarrow e_2]\!]_{\eta,\rho} &= [\![e_2]\!]_{\eta \cup \{v_1 \mapsto a, v_2 \mapsto b\},\rho} \text{ for } [\![e_1]\!]_{\eta,\rho} = (a,b) \\
[\![\mathsf{inl}(e)]\!]_{\eta,\rho} &= \mathrm{inl}\big([\![e]\!]_{\eta,\rho}\big) \\
[\![\mathsf{inr}(e)]\!]_{\eta,\rho} &= \mathrm{inr}\big([\![e]\!]_{\eta,\rho}\big) \\
\left[\!\!\left[\begin{array}{r} \mathsf{match}\ e_1\ \mathsf{with} \\ \mid \mathsf{inl}\,(v) \Rightarrow e_2 \\ \mid \mathsf{inr}(v) \Rightarrow e_3 \end{array} \right]\!\!\right]_{\eta,\rho} &= \begin{cases} [\![e_2]\!]_{\eta \cup \{v \mapsto a\},\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = \mathrm{inl}(a) \\ [\![e_3]\!]_{\eta \cup \{v \mapsto a\},\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = \mathrm{inr}(a) \end{cases} \\
[\![\mathsf{nil}]\!]_{\eta,\rho} &= [\,] \\
[\![\mathsf{cons}(e_h, e_t)]\!]_{\eta,\rho} &= [\![e_h]\!]_{\eta,\rho} :: [\![e_t]\!]_{\eta,\rho} \\
\left[\!\!\left[\begin{array}{l} \mathsf{match}\ e_1\ \mathsf{with} \\ \quad \mid \mathsf{nil} \Rightarrow e_2 \\ \quad \mid \mathsf{cons}(h,t) \Rightarrow e_3 \end{array} \right]\!\!\right]_{\eta,\rho} &= \begin{cases} [\![e_2]\!]_{\eta,\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = [\,] \\ [\![e_3]\!]_{\eta',\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = a :: l \\ & \quad \text{and } \eta' = \eta \cup \left\{ \begin{array}{c} h \mapsto a \\ t \mapsto l \end{array} \right\} \end{cases} \\
[\![\mathsf{leaf}]\!]_{\eta,\rho} &= \mathrm{leaf} \\
[\![\mathsf{node}(e_a, e_l, e_r)]\!]_{\eta,\rho} &= \mathrm{node}\big([\![e_a]\!]_{\eta,\rho}, [\![e_l]\!]_{\eta,\rho}, [\![e_r]\!]_{\eta,\rho}\big) \\
\left[\!\!\left[\begin{array}{l} \mathsf{match}\ e_1\ \mathsf{with} \\ \quad \mid \mathsf{leaf} \Rightarrow e_2 \\ \quad \mid \mathsf{node}(a,l,r) \Rightarrow e_3 \end{array} \right]\!\!\right]_{\eta,\rho} &= \begin{cases} [\![e_2]\!]_{\eta,\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = \mathrm{leaf} \\ [\![e_3]\!]_{\eta',\rho} & \text{if } [\![e_1]\!]_{\eta,\rho} = \mathrm{node}(u,v,w) \\ & \quad \text{and } \eta' = \eta \cup \left\{ \begin{array}{c} a \mapsto u \\ l \mapsto v \\ r \mapsto w \end{array} \right\} \end{cases} \\
[\![f(e_1, \ldots, e_n)]\!]_{\eta,\rho} &= \rho\big(f\big)\big([\![e_1]\!]_{\eta,\rho}, \ldots, [\![e_n]\!]_{\eta,\rho}\big)
\end{aligned}
$$

An LF program P with signature Σ, and for all $f \in \mathrm{dom}(\Sigma)$ the defining term of function f being e_f, is then interpreted as the least valuation ρ such that for all functions $f \in \mathrm{dom}(\Sigma)$ holds

$$\rho(f)(x_1, \ldots, x_n) = [\![e_f]\!]_{\eta,\rho}$$

where $\eta(v_i) = x_i$.

Program Examples

We return now to Examples of Appendix A.2 and state what we would naturally view as their translations into LF:

Example A.3.3 (Reverse – LF)**.** Again, the function `reverse` takes a list of natural numbers and reverses its order. `reverse` is implemented via the tail-recursive function `rev_aux`, which receives another list as a second argument.

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

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

Note that there is no trace remaining of the resource handling, hence we cannot easily determine how much heap space is consumed by an execution of `reverse` with arbitrary input lists.

The computation of function `reverse` works as shown below. For simplicity, we use the shorthands $\mathsf{nil} = [\,]$ and $\mathsf{cons}\big(1, \mathsf{cons}(2, \mathsf{nil})\big) = [1, 2]$ as usual.

$$\begin{aligned} \texttt{reverse}\big([1,2,4]\big) &= \texttt{rev\_aux}([1,2,4],[\,]) \\ &= \texttt{rev\_aux}([2,4],[1]) = \texttt{rev\_aux}([4],[2,1]) = \texttt{rev\_aux}([\,],[4,2,1]) \\ &= [4,2,1] \end{aligned}$$

A call to `reverse` is passed to `rev_aux` with the empty list denoted by nil as the required second argument. The function `rev_aux` distinguishes whether its first argument l is equal to the empty list or not. In the first case the call simply evaluates to the second argument acc. While in the second case the list l is split into head h and tail t, and the function call then evaluates to a recursive call to `rev_aux` with t as the first argument and h attached to acc as the second argument.

Example A.3.4 (Conway's Sequence – LF)**.** Now the first attempt to compute a step in Conway's Sequence as given in Example A.2.4 is successful in LF: Recall that the problem in Example A.2.4 was the multiple use of variables of the resource type,

violating linearity. Since we must not provide resource elements in order to construct a list in LF, this problem does not arise, and the program is linearly typable in LF:

$$
\begin{array}{l}
\texttt{verbal} : (\mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N}) \\
\texttt{vbaux} : (\mathsf{N}, \mathsf{L}(\mathsf{N})) \longrightarrow \mathsf{L}(\mathsf{N}) \\
\\
\texttt{verbal}(l) = \texttt{vbaux}(1, l) \\
\texttt{vbaux}(n, l) = \\
\quad \mathsf{match}\ l\ \mathsf{with} \\
\quad \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
\quad \mid \mathsf{cons}(c_1, t_1) \Rightarrow \\
\qquad \mathsf{match}\ t_1\ \mathsf{with} \\
\qquad \mid \mathsf{nil} \Rightarrow \mathsf{cons}\big(n, \mathsf{cons}(c_1, \mathsf{nil})\big) \\
\qquad \mid \mathsf{cons}(c_2, t_2) \Rightarrow \\
\qquad\quad \mathsf{if}\ c_1 = c_2 \\
\qquad\quad \mathsf{then}\ \texttt{vbaux}\big(n+1, \mathsf{cons}(c_2, t_2)\big) \\
\qquad\quad \mathsf{else}\ \ \mathsf{cons}\big(n, \mathsf{cons}\big(c_1, \texttt{vbaux}\big(1, \mathsf{cons}(c_2, t_2)\big)\big)\big)
\end{array}
$$

So the use of `verbal` on a list does not change its type and we are therefore allowed to use `verbal` recursively like in $\texttt{iter\_map}_{\texttt{verbal}}$, which was not possible in LFPL:

$$
\begin{array}{l}
\texttt{iter\_map}_{\texttt{verbal}}(l) = \mathsf{match}\ l\ \mathsf{with} \\
\qquad\qquad \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
\qquad\qquad \mid \mathsf{cons}(d, h, t) \Rightarrow \mathsf{cons}\big(d, h, \texttt{verbal}(\texttt{iter\_map}_{\texttt{verbal}}(t))\big)
\end{array}
$$

Constants of lists or trees may also be directly defined within LF, without relying to an external source:

$$
\begin{array}{l}
\texttt{naturals} : (\mathsf{N}) \longrightarrow \mathsf{L}(\mathsf{N}) \\
\texttt{naturals}(n) = \mathsf{if}\ (n = 0)\ \mathsf{then}\ \mathsf{nil}\ \mathsf{else}\ \mathsf{cons}\big(n, \texttt{natural}(n-1)\big)
\end{array}
$$

yielding

$$\texttt{naturals}(5) = [5, 4, 3, 2, 1]$$

So programming in LF is easier than programming in LFPL, but we cannot deduce much about the memory consumption of LF programs like we are able to do with LFPL programs. Therefore we aim at an automatic translation for LF into LFPL, whenever possible, and which warns the programmer whenever no bounds on memory consumption are deduceable in this way, so that he may avoid such memory leaks.

Index of Program Examples

References

[B&W88] Richard Bird and Philip Wadler
Introduction to Functional Programming
Prentice Hall International, 1988

[C&W00] Karl Crary and Stephanie Weirich
Resource Bound Certification
Symposium on Principles of Programming Languages, pages 184–198, 2000

[Con87] John H. Conway
The weird and wonderful chemistry of audioactive decay
Open Problems in Communication and Computation by T.M. Cover and B. Gopinath, eds., Springer, pages 173-188, 1987

[G&J00] Ewgenij Gawrilow and Michael Joswig,
Polymake: a framework for analyzing convex polytopes
contained in [K&Z00], *pages 43–74*
http://www.math.tu-berlin.de/diskregeom/polymake/doc/index.html

[H&P99] John Hughes and Lars Pareto
Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming
International Conference on Functional Programming, pages 70–81, 1999

[Hof00] Martin Hofmann
A type system for bounded space and functional in-place update
Nordic Journal of Computing, 7(4):258, Autumn 2000
http://www.tcs.informatik.uni-muenchen.de/~mhofmann/nordic.ps.gz

[I&K00] Atsushi Igarashi and Naoki Kobayashi
Garbage collection based on a linear type system
In preparation, http://citeseer.nj.nec.com/igarashi00garbage.html

[K&Z00] Gil Kalai and Günter M. Ziegler
Polytopes — Combinatorics and Computation
Birkhäuser Verlag, 2000

[M&S00] Alan Mycroft and Richard Sharp
A Statically Allocated Parallel Functional Language
ICALP 2000, Springer-Verlag LNCS, July 2000

[P&S91] Jens Palsberg and Michael I. Schwartzbach
Object-Oriented Type Inference
ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), pages 146–161, 1991

[Rit00] Eike Ritter
A calculus for resource allocation
http://www.cs.bham.ac.uk/~exr/papers.html, Autumn 2000

[Sch86] Alexander Schrijver
Theory of linear and integer programming
Wiley-Intersience, 1986

[T&B98] Mads Tofte and Lars Birkedal
A Region Inference Algorithm
Transactions on Programming Languages and Systems (TOPLAS), 20(4):734–767, Summer 1998

[T&T97] Mads Tofte and Jean-Pierre Talpin
Region-Based Memory Management
Information and Computation, 132(2):109–176, 1997

[Var91] Ilan Vardi
Computational Recreations in Mathematica
Addison-Wesley, 1991