CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# Improving Memory Consumption with FAITH - a Proof Assistant for Improvement Theory

Master's thesis in Computer science and engineering

Örjan Sunnerhagen

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

# Improving Memory Consumption with FAITH - a Proof Assistant for Improvement Theory

Örjan Sunnerhagen

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Improving Memory Consumption with FAITH

Örjan Sunnerhagen

Supervisor: David Sands, Department of Computer Science and Engineering
Examiner: John Hughes, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Improving Memory Consumption with FAITH

Örjan Sunnerhagen
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

## Abstract

In functional languages with lazy evaluation and sharing, such as Haskell, it is hard to reason about space usage. In particular, it is hard to know if a given local transformation in such a language will introduce an asymptotic space increase. Gustavsson and Sands have developed Space Improvement Theory to prove if such transformations introduced asymptotic space increase. However, since the proofs are long and complicated, Space Improvement proofs done on paper are prone to errors.

To tackle this problem, I have constructed FAITH, the proo**f** **a**ssistant for **I**mprovement **Th**eory. FAITH allows users to define transformational laws, such as those for Space Improvement, and then use those laws in proofs that transform one term to another step by step. In case of errors, FAITH provides helpful error messages. The main contributions of FAITH is to check proofs of inequational reasoning with custom transformational laws that can be applied inside arbitrary contexts.

Keywords: Computer Science, Haskell, optimization, performance, functional programming, formal verification, proof assistant, Improvement Theory, memory consumption, Space Improvement Theory

# Acknowledgements

I would like to thank my supervisor David Sands and my examiner John Hughes for their support in this project. I would also like to thank my family and friends for their encouragement during the project.

Örjan Sunnerhagen, Gothenburg, June 2021

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Functional programming is a programming style with many advantages and the lazy evaluation and call-by-need semantics of e.g. Haskell serves well for compiling and running such programs. However, the space usage of such programs is hard to reason about. To append three lists you can either write:

```
xs ++ (ys ++ zs)
```

or

```
(xs ++ ys) ++ zs
```

but which one should you use? Is one better than the other? Do they use the same amount of space? How can you be sure?

One way to find out is to use profiling. We could feed the transformation lots of different lengths of lists with different elements, such as numbers, unevaluated expressions et cetera and measure the space usage and make some kind of judgement. Using profiling however, we can never be sure if there is some special combination of lists where changing `xs ++ (ys ++ zs)` to `(xs ++ ys) ++ zs` increases the space usage with more than a constant factor. In other words, by using profiling, we can never be sure whether the transformation introduces a space leak or is *safe for space*.

Other transformations of lazy programs also come into question when developing regular programs, standard libraries or optimizing compilers. These include tail recursion, strictly accumulating parameters or the worker-wrapper transformation. Since these transformations are not all yet proven safe for space, a compiler optimization or library function that uses these transformations may introduce a space leak in some special case.

To be sure not to miss any special cases, we must turn to formal proofs. Such formal proofs were made by Gustavsson and Sands [14, 15, 16], where they proposed a theory for proofs about the space behaviour of transformations and showed space improvement for several transformations. For example, in [15, 16], they have shown that changing `xs ++ (ys ++ zs)` to `(xs ++ ys) ++ zs` may only change the stack and heap usage by a constant factor, but the converse is only true for the sum of the stack and heap. The proofs are for local improvements of programs, and can thus be applied in any program context.

The proofs are based on a set of transformational rules, derived from the space behaviour of how an abstract machine would evaluate a given program. To prove that a term is improved by another, you transform the first term to the other step by step, using these transformational rules. Since most structural changes of a program change the space behaviour somewhat, the transformational rules typically compensate for changes in space behaviour by adding or removing space gadgets.

One of these 50 transformational rules rules defined in [16], called (unfold-3) is the following:

$$\mathsf{let}\,\Gamma\{{}^{v}_{w}x = V\}\,\mathsf{in}\,\mathbf{C}[x] \underset{\sim}{\lhd\!\rhd} \mathsf{let}\,\Gamma\{{}^{v}_{w}x = V\}\,\mathsf{in}\,\mathbf{C}[{}^{v\curlyvee}V] \;\text{ if } x \in \mathbf{FV}(V).$$

The rule says that if a variable $x$ is bound to a value $V$ in a context $\mathbf{C}$, we can substitute the variable for its value if we add a space gadget ${}^{v\curlyvee}$ to $V$ to compensate for the decreased stack usage. For the rule to apply, $x$ also needs to be among the free variables of the value $V$, so $V$ is a recursive function or a constructor containing $x$, for example $y : x$. The side condition is needed because otherwise, the binding ${}^{v}_{w}x = V$ might get garbage collected earlier in the evaluation. The theory is explained in full in section 2.

Since most rules preserve or improve space behaviour by making a small change to a term, add or remove space gadgets and have side conditions, any substantial program transformation will be prone to errors that can only be discovered by those comfortable with the theory if the proofs are on paper. To ensure the integrity of proofs of improvement theory, some kind of formalization is therefore necessary.

A tool that supports reasoning about space improvement transformations has to fulfil a number of criteria. Firstly, the improvements are local, so the transformations should be valid under binding. Secondly, the rules are stated in the Barendregt variable convention [2], which states that all bound variables in any instance of a rule are distinct and disjoint from the free variables, so the tool needs to implement that correctly. Thirdly, the tool needs to support the binding constructs of not just the $\lambda$-calculus, but also recursive lets and case statements. Finally, since there are around 50 rules, and more rules might be added later, the tool has to make it easy to add new rules.

From these criteria, I have created the FAITH tool, a proo**f** **A**ssistant for **I**mprovement **Th**eory. The tool parses its rules and the proof, and checks that each transformation in the proof is correct with respect to the rules. It does this by implementing substitution that keeps the invariant that all variable names are unique and distinct from the free variables, in order to ensure correctness. Since all laws are parsed rather than implemented, the tool can be used both for Space Improvement Theory and a subset of Time Improvement Theory, as well as any other similar set of transformational rules applied to the same language.

I have used the FAITH tool to verify some of the proofs in [16, 15] in section 3, to serve as example usage. I will then explain the underlying technology in section 4.

I evaluated some other tools for constructing proofs about programming languages [1, 37, 45, 9] and a tool specifically for time improvement [23]. However, as concluded in section 5.1.1, none of these tools were suitable for space improvement. We also discuss other related work in section 5.1.2. Thereafter, we evaluate the tool by comparing it to UNIE [23] and talk about testing in section 5.2. I mention some lessons from development in section 5.3 and suggest future work in section 5.4. For a user guide, see appendix A. The tool is open-source and available at [48].

# 2
# Background - Space Improvement Theory

This section will summarize space improvement theory, as described in [14] and [16]. We will not go into many details and comment on the design or show proofs. For these details, I refer the reader to [14, 16]. Readers that are not interested in the motivation for the laws of space improvement can skip ahead to section 2.7.

## 2.1 Language

The language of space improvement theory is a small functional language. A program is a term, which we will denote as $L$, $M$ and $N$, and can be formed by the following grammar:

$$\begin{aligned} L, M, N ::= {} & x \mid \lambda x.M \mid M\,x \mid c\,\vec{x} \mid \mathsf{seq}\,M\,N \\ & \mid n \mid M + N \mid \mathsf{add}_n\,M \mid \mathsf{iszero}\,M \\ & \mid \mathsf{let}\,\{\vec{x} = \vec{M}\}\,\mathsf{in}\,N \mid \mathsf{case}\,M\,\mathsf{of}\,\{c_i \rightarrow N_i\}. \end{aligned}$$

The letter $x$ is for variables, $n$ is for numbers and $c$ is for constructors. Constructors are always fully applied and the vector notation $\vec{x}$ is short for $x_1 \dots x_n$, so $c\,\vec{x}$ is an abbreviation for $c\,x_1 \dots x_n$, where $n$ is the arity of the constructor $c$. In the same way, $\mathsf{let}\,\{\vec{x} = \vec{M}\}\,\mathsf{in}\,N$ is an abbreviation for $\mathsf{let}\,\{x_1 = M_1, \dots, x_n = M_n\}\,\mathsf{in}\,N$. The case construct $\mathsf{case}\,M\,\mathsf{of}\,\{c_i \rightarrow N_i\}$ also contains several cases, so it could be written in the longer form $\mathsf{case}\,M\,\mathsf{of}\{c_1\,\vec{x_1} \rightarrow N_1, \dots, c_n\,\vec{x_n} \rightarrow N_n\}$.

We will denote values with $V$ and $W$ and the language defines them as terms in weak normal head form, so $V, W ::= n \mid c\,\vec{x} \mid \lambda x.M$. The construct $\mathrm{seq}\,M\,N$ evaluates $M$ to a value, throws away its result and then evaluates $N$. Since $M\,x$ is in the language and $M\,N$ is not, arguments to functions must be variables. This choice is to make sure that the argument is bound to a variable on the heap, which will enable sharing, as we will see in the abstract machine later.

When $M\,x$ is evaluated, $M$ is first evaluated to $V = \lambda y.N$ and then the application is performed. The procedure is similar for $\mathsf{case}$, $\mathsf{seq}$ , $+$, $\mathsf{add}_n$ and $\mathsf{iszero}$ , so to generalize this concept, the language uses *reduction contexts.* Reduction contexts are denoted by $R$ and are defined by:

$$R ::= [\cdot]\,x \mid \mathsf{case}\,[\cdot]\,\mathsf{of}\,\{c_i\,\vec{x_i} \rightarrow N_i\} \mid \mathsf{seq}\,[\cdot]\,M \mid [\cdot] + M \mid \mathsf{add}_n\,[\cdot] \mid \mathsf{iszero}\,[\cdot]$$

In these, $[\cdot]$ is the hole of the context, so it is where the term will go. For example, if $R = [\cdot] + 2$ and $M = 1$, then $R[M] = 1 + 2$.

## 2.2 Recursive let

The let in this language is recursive. This means that you can create terms such as the mutually recursive even/odd function

```
let isEven = \x if x == 0 then True  else isOdd  (x-1)
    isOdd =  \x if x == 0 then False else isEven (x-1)
in isEven 5
```

which in the language of space improvement would be

$$
\begin{aligned}
&\mathsf{let}\,\{isEven = \lambda x.\mathsf{case}\,(\mathsf{iszero}\,x)\,\mathsf{of}\\
&\qquad\qquad\qquad\{\mathsf{true} \to \mathsf{true}\\
&\qquad\qquad\qquad,\mathsf{false} \to \mathsf{let}\,\{y = x - 1\}\,\mathsf{in}\,isOdd\,y\},\\
&\quad isOdd = \lambda x.\mathsf{case}\,(\mathsf{iszero}\,x)\,\mathsf{of}\\
&\qquad\qquad\qquad\{\mathsf{true} \to \mathsf{false}\\
&\qquad\qquad\qquad,\mathsf{false} \to \mathsf{let}\,\{y = x - 1\}\,\mathsf{in}\,isEven\,y\}\\
&\\
&\}\,\mathsf{in}\,isEven\,5
\end{aligned}
$$

if minus $(-)$ were added to the language in a similar way to $+$. Just like *isOdd* can refer to *isEven* and vice versa, it is also possible to create the infinite list of ones through $\mathsf{let}\,\{x = 1, xs = x : xs\}\,\mathsf{in}\,xs$. This means that in the construct $\mathsf{let}\,\{\vec{x} = \vec{M}\}\,\mathsf{in}\,N$, all the variables $\vec{x}$ are bound in all the expressions $\vec{M}$ and in the expression $N$.

## 2.3 Semantic Equivalence

In Space Improvement theory, semantic equivalence, which we will denote by $\equiv$, will be explained in this section. Firstly, the names of the variables do not matter, as long as they have the same binding structure. This is called $\alpha$-equivalence and is expressed in lambda calculus as

$$\lambda x.M \equiv \lambda y.M[{}^{y}\!/\!_{x}] \qquad \text{if } y \notin \mathbf{FV}(M),$$

and generalized to case-statements and lets in the obvious way. Furthermore, the order of bindings in a let or cases in a case-statement does not matter. This means that

$$\begin{aligned}
&\mathsf{let}\,\{x = 1, xs = x : xs\}\,\mathsf{in}\\
&\qquad \mathsf{case}\,xs\,\mathsf{of}\,\{\mathsf{nil} \to 2,\\
&\qquad\qquad\qquad y : ys \to 3\}\\
&\equiv\\
&\mathsf{let}\,\{as = a : as, a = 1\}\,\mathsf{in}\\
&\qquad \mathsf{case}\,as\,\mathsf{of}\,\{b : bs \to 3,\\
&\qquad\qquad\qquad \mathsf{nil} \to 2\}.
\end{aligned}$$

## 2.4 Abstract Machine

$$\begin{aligned}
\langle \Gamma\{x = M\},\, x,\, S\rangle &\to \langle \Gamma,\, M,\, \#x : S\rangle && \text{(Lookup)}\\
\langle \Gamma,\, V,\, \#x : S\rangle &\to \langle \Gamma\{x = V\},\, V,\, S\rangle && \text{(Update)}\\
\langle \Gamma,\, \mathsf{let}\,\Gamma'\,\mathsf{in}\,N,\, S\rangle &\to \langle \Gamma\Gamma',\, N,\, S\rangle && \text{(Letrec)}\\
\langle \Gamma,\, R[M],\, S\rangle &\to \langle \Gamma, M, R : S\rangle && \text{(Push)}\\
\langle \Gamma,\, V,\, R : S\rangle &\to \langle \Gamma,\, M,\, S\rangle\ \text{ if } R[V] \rightsquigarrow M && \text{(Reduce)}
\end{aligned}$$

$$\begin{aligned}
(\lambda x.M)\, y &\rightsquigarrow M[{}^{y}\!/\!_{x}]\\
\mathsf{case}\, c_j\, \vec{y}\, \mathsf{of}\, \{c_i\, \vec{x_i} \to M_i\} &\rightsquigarrow M_j[{}^{\vec{y}}\!/\!_{\vec{x_j}}]\\
\mathsf{seq}\, V\, M &\rightsquigarrow M\\
m + N &\rightsquigarrow \mathsf{add}_m\, N\\
\mathsf{add}_m\, n &\rightsquigarrow \ulcorner m + n \urcorner\\
\mathsf{iszero}\, m &\rightsquigarrow \begin{cases} \mathsf{true} & \text{if } m = 0\\ \mathsf{false} & \text{otherwise}\end{cases}
\end{aligned}$$

**Figure 2.1:** Sestoft's mark 1 abstract machine [44], used by Gustavsson and Sands' space improvement theory [16, 14]

The abstract machine of space improvement theory is shown in figure 2.1. First an explanation of the notation: A machine configuration is denoted $\langle\Gamma, M, S\rangle$, where $\Gamma$ is the heap, $M$ is the term being evaluated and $S$ is the stack. The heap is a set, while the stack is ordered, using the $R : S$ to denote that the reduction $R$ is put on top of the stack $S$. The notation $\#x$ is an update marker for $x$ and $\ulcorner m + n \urcorner$ is the mathematical addition of two numbers, as opposed to the syntactic construct of addition of two terms $M + N$.

The abstract machine expresses the lazy semantics of e.g. Haskell; arguments are only evaluated when needed (call-by-need) and the result of an evaluation may

$$
\begin{aligned}
&\langle \Gamma_1,\ \mathsf{let}\,\{y = N\}\,\mathsf{in}\,(\lambda x.M)\,y,\ S_1\rangle \\
\to&\langle \Gamma_1\{y = N\},\ (\lambda x.M)\,y,\ S_1\rangle \\
\to&\langle \Gamma_1\{y = N\},\ \lambda x.M,\ ([\cdot]\,y) : S_1\rangle \\
\to&\langle \Gamma_1\{y = N\},\ M[{}^{y}\!/\!{}_{x}],\ S_1\rangle \\
&\ \vdots \\
\to&\langle \Gamma_2\{y = N\},\ y,\ S_2\rangle \\
\to&\langle \Gamma_2,\ N,\ \#y : S_2\rangle \\
&\ \vdots \\
\to&\langle \Gamma_3,\ V,\ \#y : S_2\rangle \\
\to&\langle \Gamma_3\{y = V\},\ V,\ S_2\rangle \\
&\ \vdots \\
\to&\langle \Gamma_4\{y = V\},\ y,\ S_3\rangle \\
\to&\langle \Gamma_4,\ V,\ \#y : S_3\rangle \\
\to&\langle \Gamma_4\{y = V\},\ V,\ S_3\rangle
\end{aligned}
$$

**Figure 2.2:** The evaluation of $\mathsf{let}\,\{y = N\}\,\mathsf{in}\,(\lambda x.M)\,y$ by the abstract machine, given that $M$ contains $x$ more than once. Due to the (Lookup) and (Update) rule, the evaluation of $N$ to $V$ is only performed the first time $x$ is mentioned in $M$.

be reused (sharing). The machine does this lazy sharing through its use of variable bindings on the heap. Whenever a function is called, it has to call it with a variable rather than a general term. This means that to get the effect of $(\lambda x.M)\,N$, you must transform it to

$$\mathsf{let}\,\{y = N\}\,\mathsf{in}\,(\lambda x.M)\,y.$$

The evaluation of this term, given that $M$ contains $x$ more than once, is shown in figure 2.2. We see that the abstract machine puts $y = N$ on the heap with (Letrec) and after (Push) and (Reduce), it evaluates $M[{}^{y}\!/\!{}_{x}]$. When the value of $y$ is needed the first time, the (Lookup) rule is invoked, which takes $y = N$ off the heap, sets an update marker $\#y$ for $y$ and starts evaluating $N$. When $N$ has been evaluated to a value $V$, the (Update) rule puts the binding $y = V$ on the heap, so the next time $y$ is needed, the labor of evaluating $N$ to $V$ is not repeated. This mechanism for expressing laziness and sharing comes from Launchbury's natural semantics [27], which Sestoft transformed to small-steps semantics and revised to make his mark 1 abstract machine [44], which is essentially the abstract machine we see in figure 2.1.

To garbage collect, we remove any unnecessary bindings and update markers between each step of the abstract machine. To formalize the procedure, we say that the domain of a set of bindings $\Gamma$, denoted $\mathrm{dom}\,\Gamma$, are all the variables that $\Gamma$ binds, so if $\Gamma = \{x = M, y = N\}$, $\mathrm{dom}\,\Gamma = \{x, y\}$. Similarly, the domain of a stack, $\mathrm{dom}\,S$, are the variables that have an update marker in $S$, and $\mathrm{dom}\,\Gamma \cup \mathrm{dom}\,S = \mathrm{dom}\,(\Gamma, S)$.

For a configuration of an abstract machine $\langle\Gamma,\ M,\ S\rangle$ to be valid, $\operatorname{dom}(\Gamma, S)$ must contain all free variables of $\Gamma$, $M$ and $S$. If we can remove any bindings or update markers while still keeping the configuration valid, we do so between each step of the abstract machine. We denote this garbage collection with $\gtrdot$.

To measure the space usage, the stack and heap usage is measured at each step. When placed on the heap or stack respectively, bindings take one heap space, reduction contexts take one stack space and update markers takes one stack space *and* one heap space. The motivations for this is found in [14, 16], but in short, this ensures that the space measurement is at most a constant factor away from the actual space usage, where the constant is proportional to the size of the program.

## 2.5 Improvement

The maximum heap and stack usage, $(h, s)$, is the space used in a computation. We write $M \Downarrow_{(h,s)}$ to say that $\langle\varnothing,\ M,\ \epsilon\rangle$ is evaluated to some final state $\langle\Gamma,\ V,\ \epsilon\rangle$ with a maximum heap size at most $h$ and maximum stack size at most $s$ (with garbage collection between each step of the abstract machine). Here, $\epsilon$ denotes the empty stack.

A typical local rewrite-optimization searches a program for some pattern, such as `(xs ++ ys) ++ zs`, and replaces it with something (potentially) more effective, such as `xs ++ (ys ++ zs)` at each location where that pattern appears. We want to formalize this notion. We let $M$ be the initial term and $N$ be the better term. We can then say that there is a context $\mathbf{C}$ that contains $M$ in several places, and we take out the $M$s and leave holes that we fill with $N$ instead, turning $\mathbf{C}[M]$ into $\mathbf{C}[N]$. Not all lists have the names `xs`, `ys` and `zs`, however, so we want to substitute the variables in $M$ and $N$ for any other variables to fit the program. We let $\sigma$ range over such substitutions $[\vec{x}/\vec{y}]$, which are always variable-for-variable, and arrive at the notation that $\mathbf{C}[M\sigma]$ is changed to $\mathbf{C}[N\sigma]$.

As mentioned above, contexts are "programs with holes", which gives rise to the following definition, where we let $\mathbf{C}$ and $\mathbf{D}$ range over contexts:

$$\begin{aligned} \mathbf{C}, \mathbf{D} ::= {} & [\cdot] \mid x \mid \lambda x.\mathbf{C} \mid \mathbf{C}\,x \mid c\,\vec{x} \mid \mathsf{seq}\,\mathbf{C}\,\mathbf{D} \\ & \mid n \mid \mathbf{C} + \mathbf{D} \mid \mathsf{add}_n\,\mathbf{C} \mid \mathsf{iszero}\,\mathbf{C} \\ & \mid \mathsf{let}\,\{\vec{x} = \vec{\mathbf{C}}\}\,\mathsf{in}\,\mathbf{D} \mid \mathsf{case}\,\mathbf{C}\,\mathsf{of}\,\{c_i \to \mathbf{D}_i\}. \end{aligned}$$

Technically, contexts do not need to have holes, but in most useful cases they will.

We are now able to formally state the definition of weak and strong space improvement that [14, 16] describes. A term $M$ is said to be *weakly improved* by $N$, written $M \trianglerighteq\!\!\!\!\approx N$, if there exists a linear function $f(x) = kx + m$ such that for any context $\mathbf{C}$ and any substitution $\sigma$,

$$\mathbf{C}[M\sigma] \Downarrow_{(h,s)} \implies \mathbf{C}[N\sigma] \Downarrow_{(f(h),f(s))} .$$

A term $M$ is said to be *strongly improved* by $N$, written $M \trianglerighteq\!\!\!\!\sim N$, if, for any context $\mathbf{C}$ and any substitution $\sigma$,

$$\mathbf{C}[M\sigma] \Downarrow_{(h,s)} \implies \mathbf{C}[N\sigma] \Downarrow_{(h,s)}$$

Note that $\mathbf{C}[M\sigma] \Downarrow_{(h,s)}$ means that the heap and stack size cannot be larger than $h$ and $s$ respectively, but it may be smaller. This means that the strong improvement $M \underset{\sim}{\trianglerighteq} N$ says that $N$ cannot use more space than $M$, and the weak improvement $M \underset{\approx}{\trianglerighteq} N$ says that $N$ can use more space than $M$, but not more space than what can be bounded by a linear function.

It may seem peculiar that we would need the strong space improvement. The measurement of space is only an approximation that is a constant away from the actual space usage, so adding the linear function to define weak improvement would seem harmless. However, Gustavsson and Sands [14, 16] have shown we have to use strong improvement to make any meaningful proofs. They show that if we use strong improvement, we only have to consider the context that immediately needs to fill its holes with values to be able to generalize to all contexts. They also show that we need strong improvement to create a valid proof strategy for recursive functions. Since a strong space equivalence $\underset{\sim}{\lhd\!\rhd}$ (both $\underset{\sim}{\trianglelefteq}$ and $\underset{\sim}{\trianglerighteq}$) cannot change the space usage, not by even a constant factor (with respect to the abstract machine), transformational laws of strong space equivalence need to be more restrictive than those of weak space equivalence. This is one of the reasons that time improvement is easier than space improvement; time improvement proofs can be carried out mostly in a weak improvement, while space improvement proofs need to be carried out mostly in strong improvement.

## 2.6 Gadgets

To show weak space equivalence $M \underset{\approx}{\lhd\!\rhd} N$ (both $M \underset{\approx}{\trianglerighteq} N$ and $N \underset{\approx}{\trianglerighteq} M$), a common technique, used by Gustavsson and Sands [14, 16], is to annotate $M$ and $N$ with some gadgets to produce $M'$ and $N'$ such that $M \underset{\approx}{\lhd\!\rhd} M' \underset{\sim}{\lhd\!\rhd} N' \underset{\approx}{\lhd\!\rhd} N$. The aim of a gadget is therefore to modify the term's space usage within the allowance granted by the linear function of weak improvement. We will briefly explain these gadgets below.

### 2.6.1 Dummy references

Since the abstract machine garbage-collects bindings and update markers at each step, one way to increase space usage is to hold on to a variable longer than necessary. To do this, we use dummy references. The dummy reference is defined as

$$^{\{\vec{x}\}}M \stackrel{\text{def}}{=} \mathsf{let}\,\{\vec{y} = \vec{x}\}\,\mathsf{in}\,M \quad \text{where } \vec{y} \text{ are fresh.}$$

Since $\vec{x}$ will be part of the free variables of $^{\{\vec{x}\}}M$, they will not be garbage-collected until $^{\{\vec{x}\}}M$ is evaluated. When $^{\{\vec{x}\}}M$ is evaluated, the bindings will be immediately garbage-collected;

$$\begin{aligned}
&\langle \Gamma,\ \mathsf{let}\,\{\vec{y} = \vec{x}\}\,\mathsf{in}\,M,\ S\rangle \\
\to&\langle \Gamma\{\vec{y} = \vec{x}\},\ M,\ S\rangle \\
\gtrdot&\langle \Gamma,\ M,\ S\rangle
\end{aligned}$$

so there are no other side effects. Gustavsson and Sands [16] show that dummy references can be introduced while retaining weak space equivalence as long as they reference a *dummy binding*, i.e. a binding that is just used for taking up space;

$$\mathsf{let}\,\{x = M\}\,\mathsf{in}\,N \mathrel{\underset{\approx}{\lhd\!\rhd}} \mathsf{let}\,\{z = \Omega, x = {}^{\{z\}}M\}\,\mathsf{in}\,N \ \text{ where } z \text{ is fresh.}$$

In this equation, $\Omega$ is a divergent term. The divergent term does not terminate and the amount of stack and heap weight does not matter (${}^{w\curlyvee}\Omega \mathrel{\underset{\sim}{\lhd\!\rhd}} \Omega$ and ${}^{w\curlywedge}\Omega \mathrel{\underset{\sim}{\lhd\!\rhd}} \Omega$). It can be for example $\mathsf{let}\,x = x\,\mathsf{in}\,x$ or a type error. In this law, $\Omega$ can be replaced with anything, since it will not be used.

There are also other rules that introduce or remove dummy bindings as part of the transformation, such as (reduction);

$${}^{w}R[V] \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{w\curlyvee X}N \qquad \text{if } R[V] \rightsquigarrow N \text{ and } \mathbf{FV}(R[V]) = \mathbf{FV}({}^{X}N),$$

where the ${}^{w\curlyvee}$ and $w$ annotations will be explained in the next chapters.

### 2.6.2 Spikes

To create a momentary increase in stack- or heap size, Gustavsson and Sands [16] introduces stack- and heap spikes. Stack spikes are defined as

$${}^{\curlyvee}M \stackrel{\text{def}}{=} \mathsf{case}\,\mathsf{true}\,\mathsf{of}\,\{\mathsf{true} \to M\},$$

since the case-reduction is put on the stack for one unit of time, while $\mathsf{true}$ is evaluated.

Heap spikes are defined as

$${}^{\curlywedge}M \stackrel{\text{def}}{=} \mathsf{let}\,\{z = \Omega\}\,\mathsf{in}\,{}^{\{z\}}M \text{ where } z \text{ is fresh,}$$

since the binding $z = \Omega$ is kept on the heap by the dummy reference ${}^{\{z\}}M$ for one time unit, and is then garbage-collected in the next step, when ${}^{\{z\}}M$ is evaluated.

Gustavsson and Sands [16] show that stack- and heap spikes can be introduced while retaining weak space improvement;

$$M \mathrel{\underset{\approx}{\lhd\!\rhd}} {}^{\curlyvee}M$$

$$M \mathrel{\underset{\approx}{\lhd\!\rhd}} {}^{\curlywedge}M$$

### 2.6.3 Weights

Another way to change the space usage of a program by a constant factor is to change the amount of space a program construct uses when put on the heap or stack. This is when weights enter the picture. Firstly, we introduce a weight annotation to the $\mathsf{let}$ construct;

$$\mathsf{let}\,\{{}^{w}_{v}x = M\}\,\mathsf{in}\,N.$$

This notation means that ${}^{w}_{v}x = M$ has stack weight $w$ and heap weight $v$. This means that when the binding ${}^{w}_{v}x = M$ is put on the heap, it will take up $v$ units of space. It also means that when the update marker $\#^w_v x$ is on the stack, it will take up $w$ units of stack space and $v$ units of heap space.

Gustavsson and Sands [16] also introduce weights on reduction contexts;

$$^{w}R$$

This means that $R$ will take up $w$ units of space when placed on the stack. In summary,

$$|^w_v x = M| = (v, 0) \quad |\#^w_v x| = (v, w) \quad |^w R| = (0, w)$$

When $w$ or $v$ are not specified, they are defined to be 1, as was the case before weights were introduced.

### 2.6.4 Balloons

To set the weight of something to 0 is called introducing a balloon. Since a weight of 0 could hide other space usage, they can only be inserted in certain contexts;

$$(\lambda x.M)\, y \underset{\approx}{\lhd\!\rhd} {}^0((\lambda x.M)\, y)$$
$$\mathsf{let}\,\{x = V\}\,\mathsf{in}\,M \underset{\approx}{\lhd\!\rhd} \mathsf{let}\,\{^0x = V\}\,\mathsf{in}\,M$$

The heap weight of a binding may be set to 0 if it is a top-level definition, like a standard library definition, because it will then only be allocated once. $M \cdot x$ is syntactic sugar for ${}^0(M\,x)$

## 2.7 Language summary

In summary, the base language are *terms* that can be constructed in any of the following ways:

$$\begin{aligned} L, M, N ::= {} & x \mid \lambda x.M \mid c\,\vec{x} \mid n \mid \mathsf{let}\,\{^w_v\vec{x} = \vec{M}\}\,\mathsf{in}\,N \mid {}^wR[M] \\ & \mid {}^{\{\vec{x}\}}M \mid {}^{w\curlyvee}M \mid {}^{w\curlywedge}M \end{aligned}$$

Where $R$ is a reduction context and is defined in the following way:

$$R ::= [\cdot]\,x \mid \mathsf{case}\,[\cdot]\,\mathsf{of}\,\{c_i\,\vec{x_i} \to N_i\} \mid \mathsf{seq}\,[\cdot]\,M \mid [\cdot] + {}^wM \mid \mathsf{add}_n\,[\cdot] \mid \mathsf{iszero}\,[\cdot]$$

Values are terms in weak normal head form. The $\mathsf{let}$ construct is recursive, so in $\mathsf{let}\,\{^w_v\vec{x} = \vec{M}\}\,\mathsf{in}\,N$, the variables $\vec{x}$ are bound in $\vec{M}$ and in $N$.

There is some syntactic sugar:

- If $w$ or $v$ are not specified, they are assumed to be 1
- Patterns of the case-constuctor $c_i\,\vec{x_i}$ can be abbreviated as $pat_i$
- A set of alternatives of the case constructor $\{c_i\,\vec{x_i} \to N_i\}$ can be abbreviated as *alts*
- ${}^0(Mx)$ can be abbreviated as $M \cdot x$

$$
\begin{aligned}
\mathsf{let}\,\Gamma\{{}^{v}_{w}x = V\}\,\mathsf{in}\,\mathbf{C}[x] &\mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\,\Gamma\{{}^{v}_{w}x = V\}\,\mathsf{in}\,\mathbf{C}[{}^{v\curlyvee}V] \;\text{ if } x \in \mathbf{FV}(V) && \text{(unfold-3)}\\
\mathsf{let}\,\Gamma\{{}^{v}_{w}x = \mathsf{let}\,\Delta\,\mathsf{in}\,M\}\,\mathsf{in}\,N &\mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\,\Delta\{{}^{v}_{w}x = \mathsf{let}\,\Gamma\,\mathsf{in}\,M\}\,\mathsf{in}\,N && \text{(let-let)}\\
&\qquad\qquad \text{if}\,\mathrm{dom}\,\Gamma \cup \mathrm{dom}\,\Delta \subseteq \mathbf{FV}(M),\text{ and } |\Gamma| = |\Delta| &&\\
{}^{w}R[{}^{v}\mathsf{case}\,M\,\mathsf{of}\,\{pat_i \to N_i\}] &\mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{w+v}\mathsf{case}\,M\,\mathsf{of}\,\{pat_i \to {}^{w}R[N_i]\} && \text{(R-case)}\\
\mathsf{let}\,\Gamma\{x = V,\, y = V\}\,\mathsf{in}\,M &\mathrel{\underset{\sim}{\rhd}} \mathsf{let}\,\Gamma[{}^{x}\!/\!_{y}]\{x = V[{}^{x}\!/\!_{y}]\}\,\mathsf{in}\,M[{}^{x}\!/\!_{y}] && \text{(value-merge)}\\
\mathsf{let}\,\Gamma\{x = V,\, y = V\}\,\mathsf{in}\,M &\mathrel{\underset{\sim}{\lhd}} \mathsf{let}\,\Gamma[{}^{x}\!/\!_{y}]\{{}_{2}x = V[{}^{x}\!/\!_{y}]\}\,\mathsf{in}\,M[{}^{x}\!/\!_{y}] && \text{(value-copy)}\\
\mathsf{let}\,\{{}^{v}_{w}x = M\}\,\mathsf{in}\,\mathbf{C}[\mathsf{case}\,x\,\mathsf{of}\,\{alts, c\,\vec{y} \to \mathbf{D}[{}^{\{x\}v\curlyvee}c\,\vec{y}]\}] & && \text{(case-fold)}\\
\mathrel{\underset{\sim}{\rhd}} \mathsf{let}\,\{{}^{v}_{w}x = M\}\,\mathsf{in}\,\mathbf{C}[\mathsf{case}\,x\,\mathsf{of}\,\{alts, c\,\vec{y} \to \mathbf{D}[x]\}] & &&
\end{aligned}
$$

**Figure 2.3:** A number of laws from [16], showcasing some of the difficulty in matching, checking correctness and applying a certain rule mechanically.

- $\mathsf{let}\,x = N\,\mathsf{in}\,M\,x$ can be abbreviated as $M\,N$
- $\Gamma$ and $\Delta$ range over sets of let-bindings
- $X$ ranges over variables in dummy references, so if $X = \{\vec{x}\}$, ${}^{X}M = {}^{\{\vec{x}\}}M$

There are also contexts $\mathbf{C}$, $\mathbf{D}$. They can be constructed in the same way as terms, but at each place where there would be a term, there is a context, and a context can also be defined as the hole $[\cdot]$. A value context $\mathbf{V}$ is a context that is a value, for example $\lambda x.\mathbf{C}$. If a term $M$ is put into a context $\mathbf{C}[M]$, every hole is replaced by $M$. Free variables in $M$ might then bind to variables in $\mathbf{C}$.

There is also the divergent term $\Omega$. This term does not terminate and the amount of stack and heap weight does not matter (${}^{w\curlyvee}\Omega \mathrel{\underset{\sim}{\lhd\!\rhd}} \Omega$ and ${}^{w\curlywedge}\Omega \mathrel{\underset{\sim}{\lhd\!\rhd}} \Omega$).

The domain of a set of bindings, denoted $\mathrm{dom}\,\Gamma$, are the variables bound by it. The notation $\mathbf{FV}(M)$ denotes the free variables in $M$.

$M \mathrel{\underset{\sim}{\rhd}} N$ denotes strong improvement and $M \mathrel{\underset{\approx}{\rhd}} N$ denotes weak improvement. Strong improvement implies weak improvement.

## 2.8 Transformational Rules

Using the gadgets introduced in section 2.6 and the fact that the total space usage of a program is the maximum stack and heap usage of that program, Gustavsson and Sands [16] derive a number of laws about strong and weak space improvement (I will use laws and transformational rules interchangeably). Figure 2.3 shows some of these derived laws.

There are around 50 laws of similar nature in [16] that one has to keep in mind when proving $M \mathrel{\underset{\approx}{\rhd}} N$. You can of course always derive another rule from the abstract machine if you need a new one, so the choice of laws are always a bit ad-hoc.

We will get more into the technical details of how the FAITH tool works in section 4, but for now, let's just note some challenges with this setup. As mentioned above, the number of laws is not finite, so new laws should be easy to add if needed. The laws are defined with the Barendregt variable convention [2], which states that all bound variables in any instance of the law are distinct and they are disjoint from the free variables. So the tool needs to check that the variable convention holds for all instances of a law.

These rules show that even though the base language is quite small, the language for expressing rules is much larger. There are boolean expressions on set theory in (let-let), vectorized expressions for the branches of a case statement and weight arithmetic in (R-case). There is also substitution in (value-merge) and (value-copy).

The main difficulty however, is that some terms are repeated in the same term, such as $V$ in (unfold-3), and that you have to apply contexts correctly, as in (case-fold). This is hard because you then have to $\alpha$-rename some terms during substitution, in order to arrive at a term with unique variable names in order to uphold the Barendregt variable convention. We will explain how we solved this in section 4.

## 2.9 Induction

Gustavsson and Sands have shown that you can use induction to prove transformations of recursive terms. The idea is that if a term is recursive and terminates, it can be transformed into a nonrecursive term by unwinding the recursion a finite number of times. By the definition of Space Improvement, it only models the instances where the initial term terminates. Therefore, to show space improvement for a recursive term, you can show that space improvement for all unwindings of that term using induction. The base case is then

$$\mathsf{let}\,\{{}^{w}_{v}f = \mathbf{V}[f]\}\,\mathsf{in}\,\mathbf{C}[f^{0}] \overset{\text{def}}{=} \mathsf{let}\,\{{}^{w}_{v}f = \mathbf{V}[f]\}\,\mathsf{in}\,\mathbf{C}[{}^{\{f\}}\Omega]$$

and the induction case is

$$\mathsf{let}\,\{{}^{w}_{v}f = \mathbf{V}[f]\}\,\mathsf{in}\,\mathbf{C}[f^{n+1}] \overset{\text{def}}{=} \mathsf{let}\,\{{}^{w}_{v}f = \mathbf{V}[f]\}\,\mathsf{in}\,\mathbf{C}[{}^{w\curlyvee}\mathbf{V}[f^{n}]]$$

Gustavsson and Sands show that this only works in strong space improvement, and the proof, which is rather nontrivial, can be found in [16].

# 3 Case studies

To showcase the usability of the FAITH tool, I have used it to verify some of the transformations found in [16, 15]. These case studies were both useful for debugging FAITH, and discovered some minor errors in the proofs themselves.

All of the rules that are used in the case studies are found in figure 3.2. The FAITH representation of these rules are in figure 3.1. The full list of rules can be found in [16] and their representation in FAITH are in appendix B.1.

These case studies may seem quite small, since they only change one definition of a function to another, and are not a change to a whole program. However, one main point of Improvement Theory is that the transformations are local, so the case studies are examples of local transformations that can be applied to any program. Therefore, a larger program can have these transformations applied at any part of the program without worsening the memory consumption asymptotically. This means that optimizing compilers may apply these transformations in order to improve larger programs. More motivations for these studies are provided in [16, 15].

## 3.1 Case study 1: Cyclic structures

The first and shortest of the proofs that Gustavsson and Sands makes in [16, 15] is a proof that

$$\mathsf{let}\,\{xs = x\ :\ xs\}\,\mathsf{in}\,M$$

more space efficient than

$$\mathsf{let}\,\{repeat = \lambda x.(\mathsf{let}\,\{ys = repeat\,x\}\,\mathsf{in}\,x : ys)\}\,\mathsf{in}\,\mathsf{let}\,\{xs = repeat\,x\}\,\mathsf{in}\,M$$

The theorem is stated as: Let $\Gamma = \{repeat = \lambda x.(\mathsf{let}\,\{ys = repeat\,x\}\,\mathsf{in}\,x : ys)\}$. Then

$$\Gamma \vdash \mathsf{let}\,\{xs = repeat\,x\}\,\mathsf{in}\,M \gtrsim \mathsf{let}\,\{xs = x\ :\ xs\}\,\mathsf{in}\,M$$

```
-dummy-ref-algebra-8:              {X}d^M |~> M;
-@-rules-3:                          R[@] <~> {FV(R)}d^@;
-@-rules-2:     let G {x = {X}d^@} in N |~> let G {x = M} in N
  if FV(M) subsetof X union {x};
-spike-algebra-zero-stack-spike: [0]s^M <~> M;
-spike-algebra-zero-heap-spike:  [0]h^M <~> M;
-reduction:                    [w]^R[V] <~> [w]s^{X}d^N
  if (R[V] ~~> N) && (FV(R[V]) = FV({X}d^N));
-dummy-ref-algebra-5:               {}d^M <~> M;
-spike-algebra-13:                 [w]s^M |~> M;
-let-elim: let {x =[v,w]= M} in x <~> [w]h^M if not x in FV(M);
-let-R:                let G in [w]^R[M] <~> [w]^R[let G in M]
  if dom G subsetof FV(M);
-let-flatten:     let G1 in let G2 in M <~> let G1 G2 in M
  if dom G2 subsetof FV(M);
-unfold-5:  let G {x =[0,0]= V} in C[x]
        <~> let G {x =[0,0]= V} in C[V];
-value-merge': let G {x = let {y = V} in V} in M
           |~> let G {x = V[x/y]} in M;
```

**Figure 3.1:** The rules that are used in the case studies, in the ascii representation used in FAITH.

$$\begin{aligned}
{}^{X}M &\underset{\sim}{\rhd} M && \text{(dummy reference algebra 8)}\\
R[\Omega] &\underset{\sim}{\lhd\!\rhd} {}^{\mathbf{FV}(R)}\Omega && \text{(rule 3 for } \Omega\text{)}\\
\mathsf{let}\,\Gamma\{x = {}^{X}\Omega\}\,\mathsf{in}\,N &\underset{\sim}{\rhd} \mathsf{let}\,\Gamma\{x = M\}\,\mathsf{in}\,N && \text{(rule 2 for } \Omega\text{)}\\
&\text{if } \mathbf{FV}(M) X \cup \{x\}\\
{}^{0\curlyvee}M &\underset{\sim}{\lhd\!\rhd} M && \text{(stack algebra zero stack spike)}\\
{}^{0\curlywedge}M &\underset{\sim}{\lhd\!\rhd} M && \text{(stack algebra zero heap spike)}\\
{}^{w}R[V] &\underset{\sim}{\lhd\!\rhd} {}^{w\curlyvee X}N && \text{(reduction)}\\
&\text{if } R[V] \rightsquigarrow N \text{ and } \mathbf{FV}(R[V]) = \mathbf{FV}({}^{X}N)\\
{}^{\varnothing}M &\underset{\sim}{\lhd\!\rhd} M && \text{(dummy reference algebra 5)}\\
{}^{w\curlyvee}M &\underset{\sim}{\rhd} M && \text{(spike algebra 13)}\\
\mathsf{let}\,\{{}^{v}_{w}x = M\}\,\mathsf{in}\,x &\underset{\sim}{\lhd\!\rhd} {}^{w\curlywedge}M \quad \text{if } x \notin \mathbf{FV}(M) && \text{(let-elim)}\\
\mathsf{let}\,\Gamma\,\mathsf{in}\,{}^{w}R[M] &\underset{\sim}{\lhd\!\rhd} {}^{w}R[\mathsf{let}\,\Gamma\,\mathsf{in}\,M] \quad \text{if } \mathrm{dom}\,\Gamma \subseteq \mathbf{FV}(M) && \text{(let-R)}\\
\mathsf{let}\,\Gamma\,\mathsf{in}\,\mathsf{let}\,\Delta\,\mathsf{in}\,M &\underset{\sim}{\lhd\!\rhd} \mathsf{let}\,\Gamma\Delta\,\mathsf{in}\,M \quad \text{if } \mathrm{dom}\,\Delta \subseteq \mathbf{FV}(M) && \text{(let-flatten)}\\
\mathsf{let}\,\Gamma\{{}^{0}_{0}x = V\}\,\mathsf{in}\,\mathbf{C}[x] &\underset{\sim}{\lhd\!\rhd} \mathsf{let}\,\Gamma\{{}^{0}_{0}x = V\}\,\mathsf{in}\,\mathbf{C}[V] && \text{(unfold 5)}\\
\mathsf{let}\,\Gamma\{x = \mathsf{let}\,\{y = V\}\,\mathsf{in}\,V\}\,\mathsf{in}\,M &\underset{\sim}{\lhd\!\rhd} \mathsf{let}\,\Gamma\{x = V[{}^{x}\!/_{y}]\}\,\mathsf{in}\,M && \text{(value-merge')}
\end{aligned}$$

**Figure 3.2:** The rules that are used in the case studies, in LaTeX representation

The proof is by induction on the unwindings of *repeat*. The base case is stated as

$$\begin{aligned}
&\mathsf{let}\,\{xs = repeat^0\, x\}\,\mathsf{in}\, M\\
&\ \equiv \{\text{definition of unwindings}\}\\
&\mathsf{let}\,\{xs = {}^{\{repeat\}}\Omega\, x\}\,\mathsf{in}\, M\\
&\ \underset{\sim}{\trianglerighteq} \{\text{dummy reference algebra}\}\\
&\mathsf{let}\,\{xs = \Omega\, x\}\,\mathsf{in}\, M\\
&\ \underset{\sim}{\lhd\!\rhd} \{\text{rules for } \Omega\}\\
&\mathsf{let}\,\{xs = {}^{\{x\}}\Omega\}\,\mathsf{in}\, M\\
&\ \underset{\sim}{\trianglerighteq} \{\text{rules for } \Omega\}\\
&\mathsf{let}\,\{xs = x : xs\}\,\mathsf{in}\, M
\end{aligned}$$

and the induction case is stated as

$$\begin{aligned}
&\mathsf{let}\,\{xs = repeat^{(n+1)}\, x\}\,\mathsf{in}\, M\\
&\ \equiv \{\text{definition of unwindings}\}\\
&\mathsf{let}\,\{xs = {}^{0\curlyvee}(\lambda x.\mathsf{let}\,\{ys = repeat^n\, x\}\,\mathsf{in}\, x : ys)\, x\}\,\mathsf{in}\, M\\
&\ \underset{\sim}{\lhd\!\rhd} \{\text{spike algebra}\}\\
&\mathsf{let}\,\{xs = (\lambda x.\mathsf{let}\,\{ys = repeat^n\, x\}\,\mathsf{in}\, x : ys)\, x\}\,\mathsf{in}\, M\\
&\ \underset{\sim}{\lhd\!\rhd} \{\text{reduction}\}\\
&\mathsf{let}\,\{xs = {}^{\curlyvee}\mathsf{let}\,\{ys = repeat^n\, x\}\,\mathsf{in}\, x : ys\}\,\mathsf{in}\, M\\
&\ \underset{\sim}{\trianglerighteq} \{\text{spike algebra}\}\\
&\mathsf{let}\,\{xs = \mathsf{let}\,\{ys = repeat^n\, x\}\,\mathsf{in}\, x : ys\}\,\mathsf{in}\, M\\
&\ \underset{\sim}{\trianglerighteq} \{\text{inductive hypothesis}\}\\
&\mathsf{let}\,\{xs = \mathsf{let}\,\{ys = x : ys\}\,\mathsf{in}\, x : ys\}\,\mathsf{in}\, M\\
&\ \underset{\sim}{\trianglerighteq} \{\text{value-merge'}\}\\
&\mathsf{let}\,\{xs = x : xs\}\,\mathsf{in}\, M.
\end{aligned}$$

When verifying this proof in the FAITH tool, I could not find a rule in the spike algebra to add or remove stack spikes as suggested by the second step of the induction case. However, since it is used, and since it makes sense that adding a stack spike or heap spike with 0 weight is a strong space equivalence, we find that we have to add ${}^{0\curlyvee}M \underset{\sim}{\lhd\!\rhd} M$ and ${}^{0\curlywedge}M \underset{\sim}{\lhd\!\rhd} M$ to the spike algebra.

Since the FAITH tool doesn't implement metavariables, the proof in the FAITH tool replaces $M$ by $f \cdot x \cdot xs$. With this representation, we can still express essentially the same proof, since we can use the following derivation for any $M$:

$$
\begin{aligned}
& M \\
& \quad \underset{\sim}{\triangleleft\!\triangleright} \{\text{dummy reference algebra 5, stack algebra zero stack spike}\} \\
& {}^{0\curlyvee\varnothing}M \\
& \quad \underset{\sim}{\triangleleft\!\triangleright} \{\text{reduction}\} \\
& (\lambda x.M[{}^{x}\!/\!_{y}]) \cdot y \\
& \quad \underset{\sim}{\triangleleft\!\triangleright} \{\text{stack algebra zero heap spike}\} \\
& {}^{0\curlywedge}(\lambda x.M[{}^{x}\!/\!_{y}]) \cdot y \\
& \quad \underset{\sim}{\triangleleft\!\triangleright} \{\text{let-elim}\} \\
& (\mathsf{let}\,\{{}^{0}_{0}z = \lambda x.M[{}^{x}\!/\!_{y}]\}\,\mathsf{in}\,z) \cdot y \\
& \quad \underset{\sim}{\triangleleft\!\triangleright} \{\text{let-R}\} \\
& \mathsf{let}\,\{{}^{0}_{0}z = \lambda x.M[{}^{x}\!/\!_{y}]\}\,\mathsf{in}\,z \cdot y
\end{aligned}
$$

and then, in most cases, float the allocation of $z$ out of the way, in order to be able to apply the inductive hypothesis. One note about this derivation is that we have created a binding with zero heap weight. In [16], the authors have only shown that you can create a binding with zero heap weight by arguing that the definition is top-level.

The full FAITH proof can be found in appendix B.2.1, but to see the general structure, here is the base case:

```
bindings {
G = {repeat =[0,0]= \x1. let {zs = repeat x1} in x1:zs};
}

-- base case
proposition: G free(x f) |- let {xs = {repeat}d^@ x} in f <> x <> xs
                        |~> let {xs = x : xs} in f <> x <> xs;
proof: -simple -single{
  -dummy-ref-algebra-8
    ctx=(let G in let {xs = [.] x} in f <> x <> xs)
    X={repeat}
    M=@;
  |~> let {xs = @ x} in f <> x <> xs;
  -@-rules-3-lr
    ctx = (let G in let {xs = [.]} in f <> x <> xs)
    R=([.] x);
  <~> let {xs = {x}d^@} in f <> x <> xs;
  -@-rules-2
    ctx=(let G in [.])
    G=let {}
    x=xs
    X={x}
    N=(f <> x <> xs)
    M=(x:xs);
  |~> let {xs = x : xs} in f <> x <> xs;
} qed;
```

I designed FAITH to separate correctness from user friendliness. This is why the proof needs to be very specific, specifying all the substitutions and exactly which rule that should be used and in which direction. The reason for the design is that by parsing and checking a very detailed proof, the FAITH tool doesn't have to guess anything, and the amount of functionality and thereby the amount of code that needs to be trusted is kept to a minimum. The idea is that if the FAITH tool is developed further to incorporate guessing features such as matching, the tool could then also generate a very detailed proof such as this one, and have it be checked by the current version of FAITH. I will elaborate more on matching in section 5.4.4.

To create a proof, we start by declaring `G` $= \Gamma$ in `bindings`. In the proposition, we declare the free variables. Even though this can be inferred, FAITH lets the user define its free variables so that if there are more variables that are free in another term, FAITH does not have to guess which term it is that contain the correct free variables. Since induction is not implemented, we skip the first step of the definition of unwindings and go directly to its definition. We then complete all transformations to get to the goal.

The way that FAITH implements derivations in context is slightly different from [16]. Instead of applying a new proof technique, we treat it as syntactic sugar. That is

$$\Gamma \vdash M \mathrel{\underset{\sim}{\trianglerighteq}} N \overset{\text{def}}{=} \mathsf{let}\,\Gamma\,\mathsf{in}\,M \mathrel{\underset{\sim}{\trianglerighteq}} \mathsf{let}\,\Gamma\,\mathsf{in}\,N$$

and similar for the other improvement relations. This is why we specify `ctx=(let G in [.])` in the last step, and not `ctx= [.]`. This has also slight consequences for the induction principle; see section 5.4.1.

Because of the change from $M$ to $f \cdot x \cdot xs$, the induction case is much longer than the proof in [16]. This is because once we have gotten to

$$\mathsf{let}\,\{xs = \mathsf{let}\,\{ys = repeat^n\,x\}\,\mathsf{in}\,x : ys\}\,\mathsf{in}\,f \cdot x \cdot xs,$$

we have to get to

$$\mathsf{let}\,\{xs = \mathsf{let}\,\{{}^0_0g = \lambda a.\lambda as.a : as\}\,\mathsf{in}\,\mathsf{let}\,\{ys = repeat^n\,x\}\,\mathsf{in}\,g \cdot x \cdot ys\}\,\mathsf{in}\,f \cdot x \cdot xs$$

in order to apply the inductive hypothesis to get to

$$\mathsf{let}\,\{xs = \mathsf{let}\,\{{}^0_0g = \lambda a.\lambda as.a : as\}\,\mathsf{in}\,\mathsf{let}\,\{ys = x : ys\}\,\mathsf{in}\,g \cdot x \cdot ys\}\,\mathsf{in}\,f \cdot x \cdot xs$$

and then reapply $g$ and remove the $\mathsf{let}$ definition to get to

$$\mathsf{let}\,\{xs = \mathsf{let}\,\{ys = x : ys\}\,\mathsf{in}\,x : ys\}\,\mathsf{in}\,f \cdot x \cdot xs$$

and then continue with the last step.

The FAITH tool has not implemented induction, so the proof is divided up into three proofs; base case, induction case until the induction hypothesis and induction case after the induction hypothesis. The function *repeat* is not indexed to be $repeat^n$ or $repeat^{n+1}$ either. Se section 5.4.1 for a discussion on how induction would be implemented in FAITH.

One small error that I found in the derivation is that Gustavsson and Sands seem to have to forgotten to remove the empty dummy binding after the reduction step. Therefore, these steps should be

$$
\begin{aligned}
&\mathsf{let}\,\{xs = (\lambda x.\mathsf{let}\,\{ys = repeat^n\,x\}\,\mathsf{in}\,x : ys)\,x\}\,\mathsf{in}\,M\\
&\underset{\sim}{\lhd\!\rhd}\ \{\text{reduction}\}\\
&\mathsf{let}\,\{xs = {}^{\curlyvee\varnothing}\mathsf{let}\,\{ys = repeat^n\,x\}\,\mathsf{in}\,x : ys\}\,\mathsf{in}\,M\\
&\underset{\sim}{\lhd\!\rhd}\ \{\text{dummy reference algebra}\}\\
&\mathsf{let}\,\{xs = {}^{\curlyvee}\mathsf{let}\,\{ys = repeat^n\,x\}\,\mathsf{in}\,x : ys\}\,\mathsf{in}\,M\\
&\underset{\sim}{\rhd}\ \{\text{spike algebra}\}\\
&\mathsf{let}\,\{xs = \mathsf{let}\,\{ys = repeat^n\,x\}\,\mathsf{in}\,x : ys\}\,\mathsf{in}\,M
\end{aligned}
$$

I also found that this was sometimes forgotten in the second case study. However, since the mistake was fixed by adding a single step, the proofs are still valid.

## 3.2 Case study 2: intermediate data structures

This section will describe the formalization of part of case study 2; intermediate data structures. The proof is that the function *any* can be defined either as

```
any p xs = case xs of
             [] -> False
             (y:ys) -> p y || any p ys
```

or

```
any' p = or . map p
or = foldr (||) False
```

without any difference in the asymptotic space behaviour.

Because of time constraints, I did not formalize the base case. I did however formalize the adding of gadgets to `any`, creating `any_a` and the inductive case of transforming `any_a` to `any'_a`. The full proofs can be found in appendix B.2.2. Except that the removal of the empty dummy reference binding was sometimes skipped in this case study too, I also found that because of a slightly different desugaring, my version of the proof became a few steps shorter.

In `any_a`, there is a subterm that is

$$\mathsf{let}\,\{z = \Omega\}\,\mathsf{in}\,or \cdot (p \cdot y) \cdot {}^{\{z\}}(any_a \cdot p \cdot ys)$$

However, since this applies terms to terms, it is syntactic sugar. A readable way of desugaring the application is

$$\mathsf{let}\,\{z = \Omega, a = p \cdot y, b = {}^{\{z\}}(any_a \cdot p \cdot ys)\}\,\mathsf{in}\,or \cdot a \cdot b$$

However, if one would strictly go by the definition of desugaring, the result is a bit different. The definition of desugaring is that $M\,N$ desugars to $\mathsf{let}\,x = N\,\mathsf{in}\,M\,x$, where $x$ is fresh. This means that the subterm would desugar to

$$\mathsf{let}\,\{z = \Omega\}\,\mathsf{in}\,\mathsf{let}\,\{ds = {}^{\{z\}}(any_a \cdot p \cdot bs)\}\,\mathsf{in}\,(\mathsf{let}\,\{c = p \cdot b\}\,\mathsf{in}\,or \cdot c) \cdot ds$$

After the induction step, the corresponding subterm is

$$\mathsf{let}\,\{z = \Omega, ds = {}^{\{z\}}(any_a \cdot p \cdot bs)\}\,\mathsf{in}\,\mathsf{let}\,\{c = p \cdot b\}\,\mathsf{in}\,or \cdot c \cdot ds$$

which means that you just need one step of (let-flatten) to get to

$$\mathsf{let}\,\{z = \Omega, a = p \cdot y, b = {}^{\{z\}}(any_a \cdot p \cdot ys)\}\,\mathsf{in}\,or \cdot a \cdot b$$

but (let-R) followed by (let-flatten) to get to the mechanically desugared term.

# 4
# Implementation

This chapter will describe the implementation of FAITH. We will start with a general description of the checking process. Since the process involves handling variables, we will then move on to the choice of variable representation. In this section, we will see some motivation for an invariant of unique variable names. We will then move onto some details on how FAITH implements a substitution algorithm that respects this invariant.

## 4.1 Overview of functionality

When the user requests FAITH to verify a proof, the FAITH system first parses the law file and the proof file. It then checks that all terms have unique variable names, distinct from the variables declared to be free. We will see the motivation for this in section 4.2.1. The FAITH system also performs some other correctness checks on the law and proof file. FAITH then checks each transformation step, and that the proof starts with the specified start and ends with the specified goal.

A single step is specified in the proof file by a term, a law name, a second term and the substitutions that need to be made in order for the law to justify the transformation from the first term to the other. The law file contains laws with a law that corresponds to that law name. The FAITH system treats all laws as having an outer context in which they are applied, which is specified by the `ctx` argument. The FAITH tool checks that all substitutions are of the correct type, and then performs the substitution into both sides of the law, as well as any side conditions that the law may contain. The FAITH tool checks whether the substitutions applied to the left hand side produces a term which is equivalent to the first term, and the same for the right hand side and the second term. We will see details on the implementation of equivalence in section 4.2.2. The tool also applies the substitutions to the side conditions, evaluates them and check whether they hold. If all checks are successful, the FAITH tool tells the user that the proof is correct, and if not, it produces an error and a log file.

Each step is processed sequentially, so the time complexity is linear in the number of steps of the proof. As we will see in section 4.2.2, all permutations of all `let` statements are tried when checking for alpha equivalence at each step, which means that the time complexity is also exponential in the size of the term. However, the terms are generally quite small. Each proof step could be processed in parallel, but that would require changes to the logging system, and performance was not an issue for the proofs that I verified.

## 4.2 Representation of variables

When developing FAITH, I wanted the variable representation to support two things: matching a term to a rule, and transforming a term to another term. Even though matching remains for future work, the variable representation is chosen so that it can support matching. The matching should be done with respect to $\alpha$-equivalence and the transformation should not unintentionally capture variables. To make both of these operations easily implementable, the choice of internal representation of variables is crucial.

Take the application of (unfold-1) for example:

$$\mathsf{let}\,\Gamma\,\{{}^0_0x = V\}\,\mathsf{in}\,\mathbf{C}[x] \underset{\sim}{\lhd\!\rhd} \mathsf{let}\,\Gamma\,\{{}^0_0x = V\}\,\mathsf{in}\,\mathbf{C}[V] \qquad \text{(unfold-1)}$$

When we apply the rule left-to-right, we want to make sure that the free variables in $V$ are not captured in $\mathbf{C}$ and when we apply the rule right-to-left, we want to check that the first $V$ is $\alpha$-equivalent to the second $V$. We solve this by combining two representations:

1. Concrete variable names (i.e. `String`s) that are unique with respect to the whole term
2. Locally nameless representation

We will go over the benefits of both representations in turn. In essence, concrete unique variable names enables the user to apply arbitrary transformations to the term without unintended variable capture, while a locally nameless representation enables easy checks of $\alpha$-equivalence between terms.

### 4.2.1 Concrete unique variable names

If variable names are concrete but not unique, we might accidentally capture variables when applying transformations. For instance, non-unique variable names would allow the term

$$\mathsf{let}\,\{{}^0_0x = \lambda a.y\}\,\mathsf{in}\,(\mathsf{let}\,\{y = 5\}\,\mathsf{in}\,(x\,1) + y),$$

on which we might erroneously apply (unfold-1) to get

$$\mathsf{let}\,\{{}^0_0x = \lambda a.y\}\,\mathsf{in}\,(\mathsf{let}\,\{y = 5\}\,\mathsf{in}\,((\lambda a.y)\,1) + y).$$

But this changes the final result from $y + 5$ to $5 + 5$ because of the unintended variable capture. However, with variable names that are unique with respect to the whole term, the first term would need to be $\alpha$-renamed to

$$\mathsf{let}\,\{{}^0_0x = \lambda a.y\}\,\mathsf{in}\,(\mathsf{let}\,\{z = 5\}\,\mathsf{in}\,(x\,1) + z),$$

which would be correctly transformed to

$$\mathsf{let}\,\{{}^0_0x = \lambda a.y\}\,\mathsf{in}\,(\mathsf{let}\,\{z = 5\}\,\mathsf{in}\,((\lambda b.y)\,1) + z).$$

Note that we needed to rename the second $a$ to $b$ to make variable names globally unique.

Another way to circumvent this would be to add the side-conditions on the free variables of $V$ and the holes in $\mathbf{C}$. However, if these would need to be added to each rule by the user, these side-conditions could be confused with the free-variable-restrictions that are imposed to prevent certain values to be garbage-collected, such as $x$ in (unfold-3)

$$\mathsf{let}\,\Gamma\{{}^{v}_{w}x = V\}\,\mathsf{in}\,\mathbf{C}[x] \underset{\sim}{\triangleleft\!\triangleright} \mathsf{let}\,\Gamma\{{}^{v}_{w}x = V\}\,\mathsf{in}\,\mathbf{C}[{}^{v\curlyvee}V] \;\; \text{if}\, x \in \mathbf{FV}(V).$$

It would also be time-consuming and error-prone to add these side-conditions, both if they would be added by the user and if the FAITH project would involve generating these side-conditions automatically.

In general, the tool needs to be able to implement any transformation that can be described in the law language. When stating the laws, Gustavsson and Sands follow the standard free-variable convention, so all bound variables in a law are distinct, and they are disjoint from the free variables [16, 2]. This means that if all bound variables in a term are also distinct and disjoint from the free variables, laws can be interpreted literally.

It is not entirely uncommon to have the precondition that all variables are unique; Launchbury has that precondition for his natural semantics [27] and Gustavsson and Sands' [14] abstract machine is essentially Sestoft's Mark 1 machine [44], which is built from Launchbury's semantics. Unique variable names are also used in the Static Single Assignment form (SSA), which is a representation used in low-level machine code in e.g. Low Level Virtual Machine code (LLVM) [26] to enable compiler optimizations.

### 4.2.2 Locally nameless representation

The locally nameless representation [5] is used for checking $\alpha$-equivalence between terms with free variables. This may both be between the current term and the goal and between subterms within the same term. Take for example when we want to match (unfold-1)

$$\mathsf{let}\,\Gamma\,\{{}^{0}_{0}x = V\}\,\mathsf{in}\,\mathbf{C}[x] \underset{\sim}{\triangleleft\!\triangleright} \mathsf{let}\,\Gamma\,\{{}^{0}_{0}x = V\}\,\mathsf{in}\,\mathbf{C}[V]$$

from right to left on for example

$$\mathsf{let}\,\{{}^{0}_{0}x = \lambda y.\lambda z.x + y + z\}\,\mathsf{in}\,\lambda a.\lambda b.\lambda c.x + b + c.$$

We need to check that $\lambda y.\lambda z.x + y + z$ and $\lambda b.\lambda c.x + b + c$ are $\alpha$-equivalent, and the locally nameless representation is very well suited for this.

In locally nameless representation, the bound variables use their de Bruijn index [8], while free variables have their canonical name. The de Bruijn index of a variable is the number of $\lambda$-signs between the variable and where it is bound. This means that both $\lambda y.\lambda z.x + y + z$ and $\lambda b.\lambda c.x + b + c$ has the representation

$$\lambda\lambda(x + a_1 + a_0)$$

where $a_i$ is a variable with the de Bruijn index $i$.

The locally nameless representation works well for checking equivalence between different subterms of the same term because of the restriction on variable name uniqueness. If variables were not unique, we would be able to wrongly apply (unfold-1) to for instance

$$\mathsf{let}\ \{{}^0_0 x = \lambda y.\lambda z.x + y + z\}\ \mathsf{in}\ \lambda x.\lambda b.\lambda c.x + b + c.$$

We could also compare subterms by applying de Bruijn indexing to the whole term. However, then we would have to change the index of the free variables as they were moved past new $\lambda$-signs to get to the subterm that we want to compare with. This method would be error-prone and possibly inefficient.

To make the representation work for the whole language, we need to define the de Bruijn indexing for variables bound in lets and constructors of case-statements too. For lets, we use two indexes $i$ and $j$ and say that $e_{ij}$ means the variable that is $i$ lets away and in that let, it is the $j$th variable that is bound. This means that

$$\mathsf{let}\ \{x = 1, y = 2\}\ \mathsf{in}\ y + (\mathsf{let}\ \{z = 3 + y, a = 4\}\ \mathsf{in}\ x + y + z + a)$$

has the representation

$$\mathsf{let}\ \{1, 2\}\ \mathsf{in}\ e_{01} + \mathsf{let}\ \{3 + e_{11}, 4\}\ \mathsf{in}\ e_{10} + e_{11} + e_{00} + e_{01}.$$

For variables bound in constructors in case-statements we use a similar approach. A variable with the representation $s_{ij}$ is $i$ case-statements away. In that case-statement, $s_{ij}$ is the $j$th variable bound by the constructor in the branch where $s_{ij}$ appears. This means that

$$\begin{aligned}
&\mathsf{case}\ x\ \mathsf{of}\\
&\quad c_1\, f\, g \to f + g\\
&\quad c_2\, a\, b \to a + (\mathsf{case}\ b\ \mathsf{of}\\
&\qquad\qquad\qquad\quad c_1\, d\, e \to a + b + d + e)
\end{aligned}$$

has the representation

$$\begin{aligned}
&\mathsf{case}\ x\ \mathsf{of}\\
&\quad c_1 \to s_{00} + s_{01}\\
&\quad c_2 \to s_{00} + (\mathsf{case}\ s_{01}\ \mathsf{of}\\
&\qquad\qquad\qquad\quad c_1 \to s_{10} + s_{11} + s_{00} + s_{01}).
\end{aligned}$$

You may note that in this representation, the order of the bindings in a let and the order of constructors in a case matters, even though those orders do not matter semantically. For constructors in case-statements, we can impose an order on constructors to solve this. For lets, a strict ordering is not definable on terms like

$$\mathsf{let}\ \{a = b, b = a, d = 1, e = 1\}\ \mathsf{in}\ 1.$$

This is one of the reasons that it is inconvenient to exclusively use a locally nameless representation; since the representation is based on orders which should not matter, a term cannot have a single representation which is both invariant to $\alpha$-renaming and reordering of $\mathsf{let}$s.

To solve this, we simply try all permutations of all let bindings whenever we test for alpha equivalence. This would of course introduce a substantial toll on the performance if the terms were large. However, since the terms are small and we made sure that the generation and testing of permutations were performed lazily, it seems that the performance overhead is not significant for the proofs we performed with the help of the FAITH tool. We have also not yet introduced parallelization to improve performance because it is not yet necessary.

## 4.3 Substitution

The process of substitution was more intricate than I expected. This section will attempt to explain the problem and give an overview of how it was solved.

The transformational rules are formulated in the Barendregt variable convention [2], which says that all variables in any instance of a law are distinct, and they are disjoint from the free variables. Since a law can be applied to any subterm of the main term, or even the whole main term, this means that the variable convention always has to hold for all terms. FAITH ensures that the variable convention holds by assigning unique names for all variables, and that those names are distinct from the names of the free variables.

The transformational rules also repeats some of its term-metavariables, such as $V$ on the right hand side of (unfold-1);

$$\mathsf{let}\,\Gamma\,\{^0_0 x = V\}\,\mathsf{in}\,\mathbf{C}[x] \underset{\sim}{\Leftrightarrow} \mathsf{let}\,\Gamma\,\{^0_0 x = V\}\,\mathsf{in}\,\mathbf{C}[V].$$

This means that to apply the transformation left-to-right, we have to $\alpha$-rename $V$ when it is used the second time.

The rules also contain contexts, that may have more than one hole, where a term needs to be copied. The rules may also contain multiple copies of a context, but they can't be renamed as easily as terms, since the bound variables of contexts may bind free variables of a term that is inserted into its holes.

The FAITH system has to take all this into account, but it shouldn't rename the variables more than necessary, in order to keep user friendliness.

In the UNIE tool [23], this was for the most part solved by the use of the external framework KURE [43]. However, I could not use this approach, since it would be hard to translate the transformations into KURE rewrites in a way that is general enough to be used on any transformation that FAITH can parse.

The way that the FAITH tool solves this is that we have built a custom monad that keeps the invariant that anything that leaves it will have bound variable names that do not interfere with bound variable names of its previous terms[1]. To do this, the monad keeps the substitutions and the set of forbidden names in its internal state, and renames the bound variables in its terms as much as is needed to maintain the invariant. This also means that context application and finding the domain of a set of bindings are implemented as functions inside the monad.

[1]This is in SubstitutionMonad.hs in [48]

# 5 Discussion

## 5.1 Related Work

### 5.1.1 Using other tools for Space Improvement Theory

In this section, I will evaluate some other tools that are built for formal reasoning about programs. The section explores whether one could express space improvement in another proof assistant and get a reasonable amount of help with space improvement proofs. Ultimately, I found that out of these other approaches, a ground-up implementation in Haskell was most suitable for implementing reasoning about space improvement.

#### 5.1.1.1 Time improvement and Unie

Improvement theory was mainly developed around the year 2000, but is getting more popular. The improvement theory of time [33], which uses the same abstract machine as [14] but measures the time complexity, was used in 2014 to prove that the worker-wrapper transformation is a time improvement [18]. In 2018, Handley and Hutton [23] developed a tool for creating proofs of time improvement for the theory of [33], called UNIE (University Of Nottingham Improvement Engine). Using the tool, they were able to verify the proof of [18] and several proofs from [33].

The FAITH tool is similar, but is built for space improvement theory [14]. One option could have been to base the FAITH system upon UNIE. This would have had some benefits; some transformations and utility functions for bound and free variables could have been reused, since the language of time improvement is similar to that of space improvement. There are however some issues with further developing upon UNIE's code base.

Firstly, the tool implements its own parser and pretty printer by using Alex [30] directly instead of using a parser generator like BNFC [10]. This means that the parsing is defined by functions in Alex rather than a single grammar file. This lack of a single and modifiable grammar definition makes the language hard to precisely understand and to modify. Secondly, it does not implement alpha-equivalence. Binders can be renamed manually, but it is not possible to choose the target name. This makes it impossible to prove that $\lambda x.x \underset{\approx}{\diamond} \lambda y.y$ (where $\underset{\approx}{\diamond}$ is weak time equivalence), since $x$ can only be renamed to $a$ or $b$. Thirdly, UNIE does not include a build plan, and is mainly built upon KURE [43], which needs Haskell base version 4.8.0.0, which was released in 2015, so it currently only runs in the ghc iterpreter (ghci). KURE can be exchanged for a more general and modern framework, as suggested by

Handley in his PhD thesis [22], but changing a framework in an existing codebase is too much work for the current time constraints. Finally, UNIE implements each transformational law as Haskell-functions in the program, while FAITH parses laws from a law file. This makes FAITH more flexible, but because of this difference, many things would need to be changed in UNIE if it were to adapt to this change in approach.

#### 5.1.1.2 Higher Order Abstract Syntax in Abella and Beluga

Higher Order Abstract Syntax (HOAS) is a way to represent languages with binding. The proof assistants Abella [1] and Beluga [37] implement HOAS, so that proofs about languages can be expressed and checked in the tools using HOAS syntax. In HOAS, a recursive let where the order of the bindings matter and the number of bindings is fixed (here 3) can be represented with the type

```
tm tup3 : type.
letrec3 : (tm -> tm -> tm -> tup3)
          -> (tm -> tm -> tm -> tm)
          -> tm.
mktup3 : tm -> tm -> tm -> tup3.
```

in Beluga syntax, which is similar to the representation

$$\text{letrec}(\lambda x_1.\lambda x_2.\lambda x_3.(M_1, M_2, M_3), \lambda x_1.\lambda x_2.\lambda x_3.N)$$

There may also be a more sophisticated way that also encapsulates definitions of recursive lets that can have any number of variables. However, this encoding is far from Haskell syntax, which would confuse users who are unfamiliar with HOAS. Since the users of the FAITH tool do not define the language, they should not need to worry about its encoding when constructing space improvement proofs.

While Abella [1] may have the potential to express space improvement theory, it seems that it has not yet been used for similar tasks. Among the examples for Abella, complex binding structures such as recursive lets and case statements seem to be lacking from the languages that the proofs are made about. The closest seems to be some form of recursion in a model of the Programming language for Computable Functions (PCF) [38], shown below in Abella syntax:

```
sig pcf.
kind    tm, ty                   type.
type    rec                 ty -> (tm -> tm) -> tm.

module pcf.
eval (rec T R) V :- eval (R (rec T R)) V.
```

and some reasoning about Higher-order logic programming.

Beluga [37] focuses on automation of the variable handling of general languages, and not much on proof automation. There is some automation however, and an interactive mode called Harpoon. To check the extent of the automation, we expressed a long transitivity problem of a path through a graph. When trying to prove it using

Harpoon, it seemed that Harpoon could not recommend possible next steps or solve the problem by simple search. Based on this observation, we drew the conclusion that we can serve more automation and other helpful features to the user by implementing the FAITH tool from the ground up than to formalize space improvement in Beluga.

#### 5.1.1.3 Ott

Ott [45] is a tool for specifying programming languages. It checks that the definitions are sane, and produces code in LaTeX[12] for documentation, in Coq [3], HOL [47] and Isabelle/HOL [34] for proofs and in OCaml [28] for implementation. It also creates utility functions for renaming, insertion, et cetera. It is possible to represent the binding structure of recursive let:s in Ott;

```
term, M :: 'M_' ::=
% ...
  | let s in M  ::   :: letrec (+ bind binders(s) in M +)

bindSet, s :: 's_' ::=
  | {}        :: :: nil  (+ binders = {} +)
  | x = M : s :: :: cons (+ bind x in M +) (+ bind binders(s) in M +)
                         (+ binders = x union binders(s) +)
```

and since it exports to powerful proof assistants, it is probably possible to provide the automation needed for matching, reordering of let-bindings and proof search.

However, the representation of variables that Ott generates is fully concrete, i.e. `String`s or similar for variables. Without further restrictions, this is not sufficient for space improvement theory, since the laws are formulated in the Barendregt variable convention [2], which means that fully concrete names are only a valid representation if the variable names are unique, as seen in section 4.2. Therefore, the Ott tool is unfortunately not usable for Space Improvement Theory.

#### 5.1.1.4 Hybrid

Hybrid [9] is a package for Isabelle/HOL [34] and Coq [3]. Using de-brujin indexes, it provides support for HOAS reasoning. Martin has done a case study that formalizes evaluation, typing and subject reduction for Mini-ML [32]. Mini-ML [7] is a small lambda calculus that includes recursive lets in a similar fashion to the language of space improvement. This study gives some hope of an easy formalization of space improvement. Recent work by Zanetti on PureCake [51] has also formalized recursive lets in the HOL formalization of equivalences in a call-by-name calculus, where concrete representation of variables was used.

However, I did not further explore the possibility of formalizing space improvement in HOL or Isabelle/HOL for a couple of reasons. Firstly, HYBRID [9] uses de bruijn indexing, and as seen in section 4.2, this would not be enough to implement the transformations we aim to implement in a general manner. Secondly, the level of support is limited to that of Isabelle/HOL [34], and when this decision was made, I thought that I could have time to give more support to the user by implementing matching and other guessing techniques, which could make the proofs easier to do in FAITH than in Isabelle/HOL. Thirdly, I did not have time to explore this avenue of formalization further because of the time constraints of the project and the steep learning curve for Isabelle/HOL. Finally, I believe that the FAITH tool is faster to learn than Isabelle/HOL, which makes the formalizations more accessible.

### 5.1.2 Other approaches to similar problems

According to Handley and Hutton, they were, to the best of their knowledge, the first to support inequational reasoning for Haskell programs [23]. There are also other tools for equational reasoning [50, 17, 49, 29].

Time improvement has been used even in recent years. Schmidt-Schauß and Sabel [42] have used a semantics similar to Moran and Sands' time improvement [33] to show that common subexpression elimination is a time improvement. A theory based on Moran and Sands' time improvement [33] was also used by Hackett and Hutton [19] to show that fixed point fusion, map fusion and short cut fusion are time improvements.

Hacket and Hutton have also made a library for Liquid Haskell that can measure and reason about resource usage [20]. The library requires the user to annotate the code that is to be analyzed, by writing it in a monad and inserting where in the evaluation of the code ticks should be increased or decreased. The library can thereafter check bounds on the resource usage. Their library does not model memoization by default however, so functions have to be written to be explicitly lazy to model sharing of values.

Some recent work has been done on the space improvement of [33]. Shmidt-Schauß and Dallmeyer [40] have shown space improvement and worsening for some transformations using a variation of the theory of [14]. They have also a more recent work that adds an optimal garbage collector to their theory, which makes it robust to newer implementations of garbage collection [41]. Even though they develop a tool to count the space usage in their abstract machine, their proofs are done by hand.

Quite recently, Paraskevopoulou [36] have formally shown in CoQ that closure conversion is safe for space. Although the conversion is somewhat local, they use a vastly different model of computation, and thus, we did not build upon their work. Another example of a transformation verified by a theorem prover is [4]. In this paper, Breitner shows that the Call Arity analysis and transformation is safe in the sense that a program after this transformation does not perform more allocations during evaluation. The measure is chosen since it correlates to both time and space usage. While the proof doesn't use Sands' improvement theory [33], the proof uses Sestoft's Mark 1 abstract machine [44], which is the basis of the abstract machine of [33, 14]. However, Call Arity is a global transformation, so they did not use improvement theory, since improvement theory only covers local transformations.

Gómez-Londoño et al. [11] have developed a framework for formally proving that a CakeML program stays within a certain resource limit. Since ML is strictly evaluated, the framework is not usable for Haskell. The authors state that while proofs are now made *possible*, they are currently tedious to construct.

## 5.2 Evaluation

To evaluate the tool, we compare its usability to UNIE, and discuss possible testing potential.

### 5.2.1 Comparison to UNIE

In section 5.1.1.1, I gave the reasons why I did not further develop UNIE [23] to become FAITH, but built FAITH "ground-up". In this section, we will compare the FAITH tool to UNIE as finished products.

The main benefit of UNIE is that the user does not have to be very specific about which rule to use. For the rules UNIE implements, matching is done, even for most contexts, and few arguments have to be provided. The code is also pretty-printed using non-ascii characters for e.g. time ticks and improvement relations. The ease at which one can perform the worker-wrapper improvement proof found in [18] is impressive compared to the amount of details one has to provide for similar proofs in the FAITH tool. I will describe how matching can be added to FAITH in section 5.4. UNIE has also implemented the spacing in pretty-printing, while this is also future work for FAITH.

However, while UNIE shines in the amount of guessing it implements, FAITH shines in flexibility to accommodate more proofs. The main reason for this is that UNIE implements rules as functions, while FAITH parses its rules. Using specialized functions, UNIE has implemented around 35 transformations, while FAITH covers the 50 rules from space improvement and any rule that can be expressed with the same language constructs. Because the rules are parsed in FAITH, the documentation of what each rule does is implemented automatically and is always up-to date. In the UNIE tool, the documentation has to be provided separately, which is not done for most rules. The parsing of rules also provide more of a guarantee that the rule does what it says that it does, especially since the syntax of rules is very similar to the notation in Gustavsson and Sands' paper [16].

Neither UNIE nor FAITH provides support for induction. However, there is not much development needed to implement induction in FAITH because of the general approach to transformations that FAITH implements in order to parse rules. See section 5.4.1 for more elaboration on this.

UNIE uses a Read Print Evaluate Loop (REPL) in order to communicate with the user. While this makes it easy to experiment, UNIE does not provide undo/redo, and the user must select which subterm to transform by navigating using `left`, `right`, `rhs`, `lhs` et cetera. FAITH users, on the other hand, are editing a text file, and can therefore use the undo/redo functionality provided by their text editor. Inspired by the way Coq [3] only checks a proof file until the cursor, FAITH also has a special marker `$`, that when inserted only checks the transformations up to that marker (and warns the user of this), so that users do not have to implement the whole proof before checking that they are correct so far. In order to select the subterm to transform, the user provides the context explicitly. This selection can easily be done using copy-paste.

One of the reasons that I did not develop UNIEs code base further was the lack of documentation and that the imports were unspecific. To make FAITH better in these regards, I made sure that every imported function to every module is either prefixed or explicitly imported and provided many comments. I have also put much effort into making any errors easy to understand. FAITH does this by logging most steps so that the user can see early errors that lead to later faults, as well as implementing pretty-printing for all sub-languages used. When something is not alpha-equivalent, I have also implemented a simple function to show the smaller term where there is a difference.

### 5.2.2 Tests

In a tool that checks correctness of a proof, it is essential that the tool is correct in itself. Since I did not go the type-level-programming route, this is mainly done through testing. Currently, the program contains checks for pre- and postconditions of a function in the function itself, so when a proof is run through FAITH, the tool checks that the proof is correct and that the pre- and postconditions are true for the terms that were fed into the different functions of FAITH. This makes it less likely that an incorrect proof may go through the tool because of an internal error, because the tool may discover the internal error itself. In fact, the case studies were useful for debugging, since some bugs did trigger some of the internal checks.

There are four main avenues of correctness:

1. That the transformational laws in the law file correspond to the laws in [16]

2. That the laws expressed in the law file are sound with respect to the abstract machine

3. That the proofs in Gustavsson and Sands' case studies are correct with respect to the laws

4. That internal functions of the tool are correctly implemented, with respect to their stated pre- and postconditions.

For the first avenue, we can note that the language that the laws are specified in FAITH is very similar to the LaTeX representation in [16, 15]. This means that FAITH does implement the laws that it says that it should, given that the language constructs are correctly implemented. This means that if the laws are incorrect, the error would lie in Gustavsson and Sands' paper.

To check if a law is sound with respect to the abstract machine, we could use propositions and QuickCheck [6] combined with small extensions to FAITH. This would prevent users from inserting unsound laws into the law file. However, to give QuickCheck terms to test, we would need to generate terminating instances of the laws, and the validity of the check would depend on the diversity of the generated instances. The reason that we would need terminating instances is that to check that the instances are space equivalent without using space improvement theory, we would need to evaluate both terms and check that they have the same result and memory consumption (in strong improvement). Of course, some nontermination could be interesting, since improvement theory is a theory of untyped terms, but in practice, we would only find an error if two terms have an improvement relation between them but only one terminates, so the majority of the generated terms will need to terminate.

The generation of law instances is however an avenue for future work. The main reason is that generating arbitrary terminating terms is hard. Palka, Claessen, Russo and Hughes have generated arbitrary lambda terms [35], so such a future work could be inspired by that work, but it would also need to generate potentially recursive let bindings and case statements. Such a future work would also need to involve either matching a law to a term or generating terms for the substitutions into a law, such that these terms interact when substituted into the law.

The third avenue of correctness is what the tool checks. As seen in the case studies, I did find some minor remarks there using the FAITH tool.

The fourth avenue of correctness is addressed by the internal checks in the program. However, a better design could have been to test the correctness of the functions independently using propositions and QuickCheck [6]. This would require generation of valid laws and terms. This is less work than checking that the laws are correct, since most functions do not need that the terms that they handle do terminate or even typecheck, and some may even not require the terms to be well-typed, but they would need to have unique variable names and respect the binding structure. I hope that the documentation is enough to leave automated testing for future work. The first test could be to assert that the tool should not throw internal errors for the checks that already are in place.

## 5.3 Lessons from development

We found that there were many functions that were on the whole syntax tree, which made many functions large because of the sheer size of the language. To mitigate this, the syntax tree was reduced by the "typechecking", which removed some sugaring and other things needing different constructors because of parsing. I also developed general functions to apply a function to all subterms of a term, so that only the interesting constructs needed special code ($\lambda$, cases and lets on a function that deals with bindings for example). A more structured way to do this might have been to look into template Haskell [46] and/or lenses and traversals [25]. However, I did not have time to learn how to use these.

## 5.4 Future work

In this section, I will give suggestions on future work on the FAITH tool. In developing the FAITH tool, I have aimed for a high generality, which is why I believe that these extensions could be developed without much program restructuring.

### 5.4.1 Induction

The tool does not currently implement induction, but not much is needed to implement a simple form of it. You would need to add the language construct of indexed variables, to be the $f^n$ in

$$\mathsf{let}\,\{^w_v f = \mathbf{V}[f]\}\,\mathsf{in}\,\mathbf{C}[f^{n+1}] \stackrel{\text{def}}{=} \mathsf{let}\,\{^w_v f = \mathbf{V}[f]\}\,\mathsf{in}\,\mathbf{C}[{}^{w\curlyvee}\mathbf{V}[f^n]],$$

where the index can only be $n$ or $n+1$. To apply the induction hypothesis, you would provide the context and the substitution for the free variables. Then FAITH would do the variable-for-variable substitutions and check for alpha equivalence to the left hand side of the induction hypothesis. Variable-for-variable substitution is already implemented because it is needed when implementing $(\lambda x.M)\,y \rightsquigarrow M[y/x]$. Since FAITHs version of derivations in context is slightly different from that of [16], the induction principle will also have to be modified. Since we defined derivations in contexts as

$$\Gamma \vdash M \underset{\sim}{\trianglerighteq} N \stackrel{\text{def}}{=} \mathsf{let}\,\Gamma\,\mathsf{in}\,M \underset{\sim}{\trianglerighteq} \mathsf{let}\,\Gamma\,\mathsf{in}\,N,$$

the induction principle needs to be

$$\frac{\Gamma \vdash \mathbf{C}[f^0] \underset{\sim}{\trianglerighteq} M \qquad \Gamma \vdash \mathbf{D}[\mathbf{C}[f^n]] \underset{\sim}{\trianglerighteq} \mathbf{D}[M] \implies \Gamma \vdash \mathbf{C}[f^{n+1}] \underset{\sim}{\trianglerighteq} M}{\Gamma \vdash \mathbf{C}[f] \underset{\sim}{\trianglerighteq} M}.$$

The difference lies in that the **D** is needed, because if the induction hypothesis was

$$\Gamma \vdash \mathbf{C}[f^n] \underset{\sim}{\trianglerighteq} M \stackrel{\text{def}}{=} \mathsf{let}\,\Gamma\,\mathsf{in}\,\mathbf{C}[f^n] \underset{\sim}{\trianglerighteq} \mathsf{let}\,\Gamma\,\mathsf{in}\,M,$$

we would need to bring the binding $\Gamma$ to the place where we would need to apply the induction hypothesis.

To explain this change, we have to recall the definition of strong improvement. It says that $M \underset{\sim}{\trianglerighteq} N$ means that for any context **C** and any substitution $\sigma$,

$$\mathbf{C}[M\sigma] \Downarrow_{(h,s)} \implies \mathbf{C}[N\sigma] \Downarrow_{(h,s)}.$$

This means that if the induction hypothesis was $\Gamma \vdash \mathbf{C}[f^n] \underset{\sim}{\trianglerighteq} M$, the unsugared version would be

$$\mathbf{D}[(\mathsf{let}\,\Gamma\,\mathsf{in}\,\mathbf{C}[f^n])\sigma] \Downarrow_{(h,s)} \implies \mathbf{D}[(\mathsf{let}\,\Gamma\,\mathsf{in}\,M)\sigma] \Downarrow_{(h,s)}.$$

This means that if we find $\mathbf{C}[f^n]$ somewhere in the term, we would have to bring the top-level definitions $\Gamma$ right up to $\mathbf{C}[f^n]$ in order to apply the induction hypothesis. However, if we have the induction hypothesis be $\Gamma \vdash \mathbf{D}[\mathbf{C}[f^n]] \underset{\sim}{\trianglerighteq} \mathbf{D}[M]$, the unsugared version would be

$$\mathbf{D_1}[(\mathsf{let}\,\Gamma\,\mathsf{in}\,\mathbf{D_2}[\mathbf{C}[f^n]])\sigma] \Downarrow_{(h,s)} \implies \mathbf{D_1}[(\mathsf{let}\,\Gamma\,\mathsf{in}\,\mathbf{D_2}[M])\sigma] \Downarrow_{(h,s)},$$

where $\mathbf{D_1}$ and $\mathbf{D_2}$ are contexts. With this definition, we can let $\mathbf{D_1} = [\cdot]$ and use $\mathbf{D_2}$ to apply the induction hypothesis on a subterm.

In case study 1, this proposed definition lets us apply the induction hypothesis on

$$\begin{aligned}&\mathsf{let}\,\{\mathit{repeat} = \lambda x.(\mathsf{let}\,\{\mathit{ys} = \mathit{repeat}\,x\}\,\mathsf{in}\,x : \mathit{ys})\}\\ &\mathsf{in}\,\mathsf{let}\,\{\mathit{xs} = \mathsf{let}\,\{^0_0 g = \lambda a.\lambda \mathit{as}.a : \mathit{as}\}\,\mathsf{in}\,\mathsf{let}\,\{\mathit{ys} = \mathit{repeat}^n\,x\}\,\mathsf{in}\,g \cdot x \cdot \mathit{ys}\}\,\mathsf{in}\,f \cdot x \cdot \mathit{xs}\end{aligned}$$

with $\mathbf{D_1} = [\cdot]$,

$$\mathbf{D_2} = \mathsf{let}\,\{repeat = \ldots\}\,\mathsf{in}\,\mathsf{let}\,\{xs = \mathsf{let}\,\{{}^0_0g = \lambda a.\lambda as.a : as\}\,\mathsf{in}\,[\cdot]\}\,\mathsf{in}\,f \cdot x \cdot xs$$

to get to

$$\begin{aligned}&\mathsf{let}\,\{repeat = \lambda x.(\mathsf{let}\,\{ys = repeat\,x\}\,\mathsf{in}\,x : ys)\}\\&\mathsf{in}\,\mathsf{let}\,\{xs = \mathsf{let}\,\{{}^0_0g = \lambda a.\lambda as.a : as\}\,\mathsf{in}\,\mathsf{let}\,\{ys = x : ys\}\,\mathsf{in}\,g \cdot x \cdot ys\}\,\mathsf{in}\,f \cdot x \cdot xs,\end{aligned}$$

whereas we would need to get to

$$\begin{aligned}&\mathsf{let}\,\{xs = \mathsf{let}\,\{{}^0_0g = \lambda a.\lambda as.a : as\}\\&\qquad\qquad \mathsf{in}\,\mathsf{let}\,\{repeat = \ldots\}\,\mathsf{in}\,\mathsf{let}\,\{ys = repeat^n\}\mathsf{in}\,g \cdot x \cdot ys\}\\&\mathsf{in}\,f \cdot x \cdot xs\end{aligned}$$

to be able to apply $\Gamma \vdash \mathbf{C}[f^n] \underset{\sim}{\rhd} M$, which might not be possible.

These are quite small changes to the code, but there might also be other complications.

### 5.4.2 Further support for Time Improvement Theory

Since the thesis focused on Space Improvement Theory [16], not all constructs of the laws of Time Improvement Theory [33] are covered. To also cover the whole of time improvement, FAITH would need to implement evaluation contexts $\mathbf{E}$, the $\cong$ relation, strict contexts, more support for vectorized expressions such as $\vec{x} = \vec{\mathbf{D}}[\Omega]$, and the ability to have a side condition be $M \underset{\sim}{\rhd}^{\checkmark} M$.

### 5.4.3 Further parsing capabilities

If more of the logic of space improvement would be parsed, and thus externalized and made changeable by the user, the tool could verify more kinds of proofs.

One example is that you may want to add other kind of proof techniques than induction. For example, you might want to prove $M \underset{\sim}{\lhd\!\rhd} N$ by proving $M \underset{\sim}{\rhd} N$ and then $N \underset{\sim}{\rhd} M$. These proof techniques could be specified in the law file in a syntax such as

```
-simple: -single{M ... N} |- M ... N;
-fix-point-induction:
    -base{C[f^0] |~> M}
    -induction{C[f^n] |~> M |- C[f^(n+1)] |~> M}
    |- C[f] |~> M;
-both-ways: -lr{M |~> ... N} -rl{N |~> ... M} |- M <~> N;
```

and then parsed and used in the proof, if the tool would be extended to allow such generality. The only thing to change would be the global conditions, i.e. the improvement relation and the start and goal.

Another example could be that the improvement relations ($\underset{\approx}{\lhd\!\rhd}$, $\underset{\approx}{\lhd}$, $\underset{\approx}{\rhd}$,$\underset{\sim}{\lhd\!\rhd}$,$\underset{\sim}{\lhd}$,$\underset{\sim}{\rhd}$) and their hierachy could be parsed in the law file. This would externalize more of the theory, so that it is only the language that is truly specific to improvement theory. The reduction relation could also be parsed instead of implemented. To make the tool future-proof for such generalizations, we have put the implementation of reduction and comparison of improvement relations into a separate file[1].

However, the language of improvement theory, with its binding semantics and syntax, is not something I would recommend to externalize. As exemplified by the tools that I have evaluated in section 5.1.1, it is hard to create a tool that both allow users to create readable proofs of improvement theory, while also letting the user define the language. However, since the language is well-defined and parsed using the external BNFC tool [10], extensions and/or changes to the language can be added if the source code is modified.

### 5.4.4 Matching

The main avenue for future work on FAITH is matching. Specifically, it would be to match laws on the term. Take this transformation, for example:

```
<~> let {xs = (\x2. let {ys = repeat x2} in x : ys) x}
    in f <> x <> xs;
-reduction-lr
  ctx=(let G in let {xs = [.]} in f <> x <> xs)
  w=1
  R=([.] x)
  V=(\x2. let {ys = repeat x2} in x : ys)
  X={}
  N=(let {ys' = repeat x} in x : ys');
<~> let {xs = s^{}d^(let {ys = repeat x} in x : ys)}
    in f <> x <> xs;
```

If matching was done, the user could be less specific about the proof. For example, it could be enough to specify the first and the second term, `-reduction` or a subset of the substitutions, and have the tool guess the rest. If this would be implemented, FAITH could then pretty-print the detailed proof that would be internally generated. After that, the tool could parse the pretty-printed proof, and check that it is correct using the functionality that is currently available. Using this separation, FAITH would become more user-friendly, while still producing correct proofs. Furthermore, the amount of code to trust in order to trust the correctness of the proofs would remain the same.

The problem of matching would be to match the second-order language of laws on the first-order terms. The law language is second-order because it includes contexts, which have the type $term \to term$. One way to solve this is to solve the harder problem of second-order unification. To do this, one would need to apply the context-hole variable approach used in some of the proofs in [16] and further explained in [39]. This would make contexts into functions, since the theory models capturing. After this transformation, one can apply the second-order unification algorithm of Huet et al [24] in order to match laws on contexts. However, I suspect

[1]This is LanguageLogic.hs in [48]

that the problem need not be that complicated. Another approach is to use first-order unification [31] to match the first-order law terms ($M$, $\Gamma$, $V$ et cetera). Then for the second-order patterns $\mathbf{C}$[<pattern>], search for every instance of <pattern> in the term that you match on. This would terminate, since the term you match on is fully concrete, because all the terms mentioned in the proofs need to be fully concrete.

If matching is implemented, the few terms that have to be provided could also have wildcards, so that the user doesn't have to specify the whole term. Matching would also be the first step toward proof search.

### 5.4.5 Pretty-printing

Currently, the pretty-printer prints the term on a single line. This is because the pretty-printer FAITH uses does not support the spacing conventions of functional languages such as Haskell. In particular, FAITH uses the pretty-printer from BNFC [10], which uses spacing that is built for Java-like languages. Therefore, FAITH just removes the newlines and some of the spaces. If matching is added to FAITH as described in section 5.4.4, the spacing on the detailed proof will be crucial for readability. You would then also need to test that if a term is pretty-printed and then parsed, you would arrive at an equivalent term.

## 5.5 Conclusion

In this master's thesis, I have tackled the problem that proofs of space improvement [16] are hard to construct in an easy and correct manner. To address this problem, I have constructed FAITH, the proof assistant for Improvement Theory. The tool parses its laws, and allows the user to create transformational proofs between different terms in order to create proofs of space improvement. The user has to provide unique variable names to all binding variables, provide all substitutions, and divide the proof into its non-inductive parts. However, for these inputs, FAITH will then check the correctness of the proof, and if there are any errors, provide detailed log messages to indicate the errors. FAITH also provides functionality for checking partly incomplete proofs. Since FAITH parses its laws rather than implements them as functions, the tool can be used for any set of transformations that use the same language, for example a subset of time improvement theory [33] or new laws of space improvement. Since FAITH uses unique variable names, transformations can be specified under the Barendregt variable convention [2], and the proofs that FAITH supports include transformations that are done in a binding context. I hope that the FAITH will be useful for further use of space improvement theory [16], and for verifying other transformational proofs of functional languages.

# Bibliography

[1] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

[2] Hendrik P Barendregt et al. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1.* PhD thesis, Inria, 1997.

[4] Joachim Breitner. Formally proving a compiler transformation safe. *ACM SIGPLAN Notices*, 50(12):35–46, 2015.

[5] Arthur Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49(3):363–408, 2012.

[6] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[7] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: Mini-ml. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 13–27, 1986.

[8] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[9] Amy Felty and Alberto Momigliano. Hybrid. *Journal of automated reasoning*, 48(1):43–105, 2012.

[10] Markus Forsberg and Aarne Ranta. Bnf converter. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 94–95, 2004.

[11] Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O Myreen, and Yong Kiam Tan. Do you have space for dessert? a verified space cost semantics for cakeml programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[12] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LATEX companion*, volume 1. Addison-Wesley Reading, 1994.

[13] Jörgen Gustavsson. *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages*. PhD thesis, Göteborgs Universitet, 2001. Paper I.

[14] Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. *Electronic Notes in Theoretical Computer Science*, 26:69–86, 1999.

[15] Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. *ACM SIGPLAN Notices*, 36(10):265–276, 2001.

[16] Jörgen Gustavsson and David Sands. Space safe transformations of call-by-need programs. Paper I in [13], 2001.

[17] Walter Guttmann, Helmuth Partsch, Wolfram Schulte, and Ton Vullinghs. Tool support for the interactive derivation of formally correct functional programs. *J. UCS*, 9(2):173, 2003.

[18] Jennifer Hackett and Graham Hutton. Worker/wrapper/makes it/faster. *ACM SIGPLAN Notices*, 49(9):95–107, 2014.

[19] Jennifer Hackett and Graham Hutton. Parametric polymorphism and operational improvement. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–24, 2018.

[20] Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: Reasoning about resource usage in liquid haskell. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[21] Martin AT Handley. *Efficiency three ways: tested, verified, and formalised*. PhD thesis, University of Nottingham, 2020.

[22] Martin AT Handley. Improving haskell. chapter 4 in [21], 2020.

[23] Martin AT Handley and Graham Hutton. Improving haskell. In *International Symposium on Trends in Functional Programming*, pages 114–135. Springer, 2018.

[24] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta informatica*, 11(1):31–55, 1978.

[25] Csongor Kiss, Matthew Pickering, and Nicolas Wu. Generic deriving of generic traversals. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, 2018.

[26] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[27] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154, 1993.

[28] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 3.12. *Institut National de Recherche en Informatique et en Automatique*, 2011.

[29] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 27–38, 2003.

[30] Simon Marlow. Alex. `https://www.haskell.org/alex/`, 2021 (accessed March 24, 2021).

[31] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[32] Alan J. Martin. Reasoning using higher-order abstract syntax in a higher-order logic proof environment : improvements to hybrid and a case study, 2011.

[33] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 43–56, 1999.

[34] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[35] Michał H Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 91–97, 2011.

[36] Zoe Paraskevopoulou and Andrew W Appel. Closure conversion is safe for space. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[37] Brigitte Pientka and Andrew Cave. Inductive beluga: Programming proofs. In *International Conference on Automated Deduction*, pages 272–281. Springer, 2015.

[38] Gordon D. Plotkin. Lcf considered as a programming language. *Theoretical computer science*, 5(3):223–255, 1977.

[39] David Sands. Computing with contexts a simple approach. *Electronic Notes in Theoretical Computer Science*, 10:134–149, 1998.

[40] Manfred Schmidt-Schauß and Nils Dallmeyer. Space improvements and equivalences in a functional core language. *arXiv preprint arXiv:1802.06498*, 2018.

[41] Manfred Schmidt-Schauß and Nils Dallmeyer. *Space Improvements for Total Garbage Collection.* Institut für Informatik, Johann Wolfgang Goethe-Universität, 2019.

[42] Manfred Schmidt-Schauß and David Sabel. Improvements in a functional core language with call-by-need operational semantics. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 220–231, 2015.

[43] Neil Sculthorpe, Nicolas Frisby, and Andy Gill. The kansas university rewrite engine: A haskell-embedded strategic programming language with custom closed universes. *Journal of Functional Programming*, 24(4):434–473, 2014.

[44] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[45] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, et al. Ott: Effective tool support for the working semanticist. *Journal of functional programming*, 20(1):71, 2010.

[46] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, 2002.

[47] Konrad Slind and Michael Norrish. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[48] Orjan Sunnerhagen. Proof assistant for improvement theory (faith). `https://github.com/orjansu/FAITH`, 2021 (accessed May 31, 2021).

[49] Simon Thompson and Huiqing Li. Refactoring tools for functional languages. *Journal of Functional Programming*, 23(3):293, 2013.

[50] Mark Anders Tullsen and Paul Hudak. *Path, a program transformation system for haskell.* Yale University, 2002.

[51] RICCARDO ZANETTI. Purecake: Towards a formally verified non-strict language compiler, 2020.

# A

# User Guide

## A.1 Syntax

Since it is cumbersome to work with LaTeX when using text editors and in parsing, the language is translated to ascii. There are two, slightly different languages; one for terms in the proof language, and one for terms in the law language. The proof language is used for the terms in the proofs, while the law language is used for stating laws. One reason for this distinction is that the law language uses metavariables, and would be ambigous if any names could be used for any variables. For example, we know that $c\,x\,y$ is the constructor $c$ applied to the variables $x$ and $y$ and not a variable called $c$, because we use $c$ to range over constructors, but this wouldn't be practical when writing big terms. Table A.1 shows some of the different constructs in the different languages. To indicate something recursive, I will use for example <term> to indicate a term, but < or > is not part of the language when not in `verbatim`. Here is an example of a proof file:

```
-- single line comments
{- Multiple line
   Comments
-}
bindings{
-- Let-bindings for derivations in context
A = { false = False
    , head = \xs . case xs of {y : ys -> y}};
-- Constructor declaration
Pair (2);
Triple (3);
Nothing (0);
}

proposition: A free(a b) |- s^(\c. a + b + c) |~> (\c. a + b + c);
proof: -simple -single{
  -spike-algebra-13
    ctx = (let A in [.])
    M=(\c. a + b + c)
    w=1;
  |~>
  -- note that we have to supply the |~>, but not the first and
  -- last term if we don't want to.
}qed;

-- There may be multiple proofs in a single file
proposition: A free(x) |- [1]h^[2]h^x <~> [2]h^[1]h^x;
proof: -simple -single{
  -spike-algebra-12-lr
    ctx =(let A in [.]) w=1 v=2 M=x;
  <~>
}qed;
```

Here are some syntax notes:

- All variable names of bound variables have to be unique with respect to the whole term, and these variable names have to be distinct from the free variables.
- You may rename variables and change the order of bindings in let statements or bindings in case statements as you please. If you want to do it in a separate step (for esthetic reasons), you can use the command `-alpha-equiv` without arguments.
- The improvement relations $\underset{\approx}{\Leftrightarrow}\underset{\approx}{\lhd}\underset{\approx}{\rhd}\underset{\sim}{\Leftrightarrow}\underset{\sim}{\lhd}\underset{\sim}{\rhd}\overset{\text{def}}{=}$ are `<~~>` `<~~|` `|~~>` `<~>` `<~|` `|~>` `=def=`.
- Lists of variables are separated by spaces and not commas.
- To add a stack weight of for example 3 to a reduction, add `[3]` before it, for example `[3] case a of {}`, `[3] seq a b`, `[3] (a + [3] b)` and `[3] (f x)`. You must have parenthesises to add weight to an application, like `[3] (f x)` or the outer weight of `+`, like `[3] (a + b)` or `[3] (a + [3] b)`, but you must not have a parenthesis to add weight to other reductions (`case`, `add`, `iszero` or`seq`. If you add weight to a reduction in the law language, use the same syntax as you would in the proof language, but if you want to add weight to a general reduction, the syntax is `[w] ^ R[M]` and not `[w] R[M]`.
- For parsing reasons,to highlight the fact that the bindings in a `let` and the branches in `case` are sets, and to make the law language similar to the term language, the bindings of a `let` statement and the branches of a `case` statement are surrounded by `{}` and interspersed by `,`. Therefore, the syntax is **not** the Haskell syntax

  ```
  let a = case x of
            True -> 1
            False -> 2
      b = 2
  in a + b
  ```

  but instead, the correct syntax is

```
let { a = case x of
            { True -> 1
            , False -> 2}
    , b = 2}
in a + b
```

- When specifying some substitutions, you need to add a keyword before the value to indicate its type. See table A.2 for details.
- If you want to specify a substitution to be a context that is just a hole, for example $C = [\cdot]$, you have to write `C= [.]` and not `C=[.]`.
- When specifying substitutions to a term, you will often need parenthesises around it.
- When specifying a law that has the relation `<~>`, `<~~>` or `=def=`, you will need to add the suffix `-lr` or `-rl` to indicate if you want to use the law from left to right or from right to left.
- In law language, constructors are denoted `c ys`. In proof language, they are identifiers with a capital letter, and its arguments are surrounded by a parenthesis, for example `Pair (a b)`. To be able to use a custom constructor in proofs, it must be declared in the `bindings{}` with the number of arguments. For example, the `Pair` constructor needs to be declared as `Pair (2)`
- the Cons constructor in case statements does not have parenthesises around it.

Most of these intricacies exist because the language needs to be unambigous and parseable by space-insensitive LR parsing, as implemented by BNFC [10]. For further documentation of syntax, look at the BNFC documentation of the languages in the repository [48]. For further examples of law syntax, see the laws in section 3.1. You can also compare the full set of laws in B.1 with the laws in [16]. For example proofs, see appendix B.

The syntax of transformational proofs is a list in the following order:

1. A transformational command and the substitutions, followed by `;`
2. An improvement relation (`<~~>`, `<~>`, `=def=`, `|~>` or `|~~>`)
3. A term, followed by `;`
4. Optionally, a marker `$` to indicate that the proof is only done until this point.
5. repeat from step 1

**Table A.1:** Table of some of the different language constructs in the different language representations

| Name | LaTeX | Law language | Proof Language |
|---|---|---|---|
| variables | $x, y, f, z$ | `x, y, f, z` | lower-case identifiers |
| Lambda abstraction | $\lambda x.M$ | `\ x . M` | `\` <variable> `.` <term> |
| Stack spike(s) | $^{\curlyvee}$ or $^{v\curlyvee}$ | `s^` or `[v]s^` | [<integer>]`s^` or `s^` |
| Heap spike(s) | $^{\curlywedge}$ or $^{v\curlywedge}$ | `h^` or `[v]h^` | [<integer>]`h^` or `h^` |
| Dummy references | $^{X}$ | `{X}d^` | `{`<variables>`}d^` |
| Balloon application | $x \cdot y$ | `x <> y` | `x <> y` |

**Table A.2:** Substitutions that need keywords to specify their type. If the substitution is not of this type, specify it without a keyword.

| Type | keyword | example |
|---|---|---|
| let bindings | `let` | `G=let {a = 1, b = 2}` |
| vector of variables | `variables` | `ys = variables [a b c]` |
| patterns | `patterns` | `pat_i = patterns [ [], a:as]` |
| case branches | `case` | `alts = case { [] -> False}` |
| terms | `terms` | `N_i = terms [ 0, a + as]` |

## A.2 Forbidden Law constructs

There are some law constructs that are accepted by the parser, but are not supported. Their behaviour if they are inserted is undefined. Because of time constraints, FAITH does not currently check that these constraints hold for the laws that it parses. However, note that if a law was written in a way that breaks one of the first two constraints, it cannot be instantiated, since all terms to instantiate it on would have unique variable names.

1. The metavariable for a set of bindings may not be copied in the same term, because that would imply that the same names for bound variables are used twice. For example, the law term `let G in let G in M` is forbidden.

2. Metavariables in binding positions may not be repeated, as that would mean that a variable name is repeated. For example, the law terms `let {x = M, x = N} in V`, `case M of { x : x -> N}` and `\x. \x. M` are forbidden.

3. The case statements may only be in one of the following forms:

   (a) `{alts,` <list of concrete statements>`}`

   (b) `{pat_i ->` <term> `}`, where <term> may contain `N_i`

   (c) `{` <list of concrete statements> `}`

4. There may not be multiple vectorized expressions, i.e. if you add a new vectorized expession metavariable to the `.cf` file, such as `N_j`, things might not work properly.

# B

# Detailed FAITH scripts and encodings

## B.1 All laws

These are all laws that are part of space improvement from Gustavsson and Sands [16]. Note that all relations involving `=def=`, `<~>` and `<~~>` needs the suffix `-lr` or `-rl` to indicate if you mean the left-to-right or right-to-left direction. The reduction `~~>` is implemented in code rather than rules. It is in `LanguageLogic.hs` in [48].

```
-reduction: [w]^R[V] <~> [w]s^{X}d^N
  if (R[V] ~~> N) && (FV(R[V]) = FV({X}d^N));

-unfold-1: let G {x =[v,w]= V} in C[x]
       |~> let G {x =[v,w]= V} in C[{x}d^V];
-unfold-2: let G {x =[v,w]= V} in C[x]
       <~| let G {x =[v,w]= V} in C[{x}d^[v]s^V];
-unfold-3: let G {x =[v,w]= V} in C[x]
       <~> let G {x =[v,w]= V} in C[[v]s^V]  if x in FV(V);
-unfold-4: let G {x =[0,w]= V} in C[x]
       <~> let G {x =[0,w]= V} in C[{x}d^V];
-unfold-5: let G {x =[0,0]= V} in C[x]
       <~> let G {x =[0,0]= V} in C[V];

-let-elim: let {x =[v,w]= M} in x <~> [w]h^M if not x in FV(M);
  -- Side condition is a bug I found.
-let-R: let G in [w]^R[M] <~> [w]^R[let G in M]
  if dom G subsetof FV(M);

-let-flatten: let G1 in let G2 in M <~> let G1 G2 in M
  if dom G2 subsetof FV(M);

-let-let: let G1 {x =[v,w]= let G2 in M} in N
      <~> let G2 {x =[v,w]= let G1 in M} in N
  if dom G1 union dom G2 subsetof FV(M) && |G1| = |G2|;

-let-alts: let G1 in [w] case M of {pat_i -> let G2 in N_i}
       <~> let G2 in [w] case M of {pat_i -> let G1 in N_i}
  if dom G1 union dom G2 subsetof FV(N_i) && |G1| = |G2|;

-let-let': let G1 {x =[v,w]= M} in N
       <~> let G2 {x =[v,w]= {dom G2}d^(let G1 in M)} in N
  if dom G1 subsetof FV(M) && |G1| = |G2|;

-let-alts': let G1
            in [w] case M of {pat_i -> N_i}
        <~> let G2
            in [w] case M of {pat_i -> {dom G2}d^(let G1 in N_i)}
  if dom G1 subsetof FV(N_i) && |G1| = |G2|;

-value-merge: let G {x = V, y = V} in M
          |~> let G[x/y] {x = V[x/y]} in M[x/y];

-value-copy:  let G {x = V, y = V} in M
          <~| let G[x/y] {x =[1,2]= V[x/y]} in M[x/y];

-value-merge': let G {x = let {y = V} in V} in M
           |~> let G {x = V[x/y]} in M;

-value-copy': let G1 {x = let {y = V} in V} in M
          <~| let G {x =[1,2]= V [x/y]} in M;

-gc: let G1 G2 in M <~> {X}d^(let G1 in M)
      if FV(let G1 G2 in M) = FV({X}d^(let G1 in M));

-empty-let: let {} in N <~> N;

-R-case: [w]^R[[v] case M of {pat_i -> N_i}]
     <~> [w+v] case M of {pat_i -> [w]^R[N_i]} ;

-case-unfold: case x of {alts, c ys -> D[x]}
          |~> case x of {alts, c ys -> D[{x}d^c ys]};

-case-fold: let{x =[v,w]= M}
            in C[case x of {alts, c ys -> D[x]}]
        <~| let{x =[v,w]= M}
            in C[case x of {alts, c ys -> D[{x}d^[v]s^c ys]}];

-@-rules-1: {X}d^@ |~> M            if FV(M) subsetof X;
-@-rules-2: let G {x = {X}d^@} in N |~> let G {x = M} in N
      if FV(M) subsetof X union {x};
-@-rules-3: R[@] <~> {FV(R)}d^@;
-@-rules-4: let G in {X}d^@ <~> {Y}d^@ if Y = FV(let G in {X}d^@);
-@-rules-5: [w]s^@ <~> @;
```

```
-@-rules-6: [w]h^@ <~> @;
-@-rules-7: let G {x =[v,w]= @} in C[x]
        <~> let G {x =[v,w]= @} in C[{x}d^@];

-spike-algebra-1:    [w]^R[[v]s^M] <~> [w+v]s^[w]^R[M];
-spike-algebra-2:    [w]^R[[v]h^M] <~> [v]h^[w]^R[M];
-spike-algebra-3:  let G in [v]s^M <~> [v]s^(let G in M)
  if dom G subsetof FV(M);
-spike-algebra-4:  let G in [v]h^M <~>  [v+|G|]h^(let G in M)
  if dom G subsetof FV(M);
-spike-algebra-5: [w] case M of {pat_i -> [w]s^N_i}
              <~> [w] case M of {pat_i -> N_i};
-spike-algebra-6:      [w]s^[v]s^M <~> [v]s^M if w =< v;
-spike-algebra-7:      [w]h^[v]h^M <~> [v]h^M if w =< v;
-spike-algebra-8: [w]s^{X}d^[w]s^M <~> [w]s^{X}d^M;
-spike-algebra-9: [w]h^{X}d^[w]h^M <~> [w]h^{X}d^M;
-spike-algebra-10:     [w]h^[v]s^M <~> [v]s^[w]h^M;
-spike-algebra-11:     [w]s^[v]s^M <~> [v]s^[w]s^M;
-spike-algebra-12:     [w]h^[v]h^M <~> [v]h^[w]h^M;
-spike-algebra-13:          [w]s^M |~> M;
-spike-algebra-14:          [w]h^M |~> M;


-dummy-ref-algebra-1:   [w]^R[{X}d^M] <~> [w]s^{X}d^[w]^R[M];
-dummy-ref-algebra-2: let G in {X}d^M <~> [|G|]h^{X}d^(let G in M)
                                          if dom G subsetof FV(M);
-dummy-ref-algebra-3: let G {x =[w,0]=V} in C[{x}d^M]
                  <~> let G {x =[w,0]=V} in C[{FV(V)\\{x}}d^M];
-dummy-ref-algebra-4: [w]^R[{X}d^M] <~> [w]^R[M] if X subsetof FV(R);
-dummy-ref-algebra-5:          {}d^M <~> M;
-dummy-ref-algebra-6:  {X union Y}d^M <~> {X}d^M if Y subsetof FV(M);
-dummy-ref-algebra-7:     {X}d^{Y}d^M <~> {X union Y}d^M;
-dummy-ref-algebra-8:          {X}d^M |~> M;

-dummy-ref-def: {xs}d^M =def= let {ys = xs} in M if ys are fresh;
-dummy-bind-intro: let {x = M} in N <~~> let {z = @, x = {z}d^M} in N
  if z is fresh;
-stack-spike-def: [w]s^M =def= [w] case True of {True -> M};
-heap-spike-def:  [w]h^M =def= let {x=[1,w]= @} in {x}d^M
  if x is fresh;
-stack-spike-intro: M <~~> [w]s^M;
-heap-spike-intro:  M <~~> [w]h^M;

-lemma-5-5-1: [w]^R[M] |~> [w1]^R[M] if w >= w1;
-lemma-5-5-2: [w] (M + [v] N) |~> [w1] (M + [v1] N)
  if w >= w1 && v >= v1;
```

```
-lemma-5-5-3: let G {x =[w,v]=M} in N |~> let G{ x =[w1,v1]= M } in N
  if w >= w1 && v >= v1;

-lemma-5-6-1: R[M] <~~> [w]^R[M] if w > 0;
-lemma-5-6-2: (M + N) <~~> [w](M + [v] N) if w > 0 && v > 0;
-lemma-5-6-3: let G {x = M} in N <~~> let G {x =[w,v]= M} in N
  if v > 0 && w > 0;

-balloon-intro-1: (\x.M) y <~~> (\x.M) <> y;
-balloon-intro-2: let {x = V} in N <~~> let {x =[0,1]= V} in N;
-balloon-reduction: (\x.M) <> y <~> M [y/x]   if y in FV(M[y/x]);

-unfold-weak: let G {x=[v,w]=V} in C[x]
         <~~> let G {x=[v,w]=V} in C[{x}d^V];

-- Laws that I added that I think are needed and valid
-balloon-intro-untyped: M x <~~> M <> x;
-spike-algebra-zero-stack-spike: [0]s^M <~> M;
-spike-algebra-zero-heap-spike: [0]h^M <~> M;
```

## B.2 Full FAITH proofs

### B.2.1 Case study 1, cyclic structures

```
bindings {
G = {repeat =[0,0]= \x1. let {zs = repeat x1} in x1:zs};
}

-- base case
proposition: G free(x f) |- let {xs = {repeat}d^@ x} in f <> x <> xs
                        |~> let {xs = x : xs} in f <> x <> xs;
proof: -simple -single{
  -dummy-ref-algebra-8
    ctx=(let G in let {xs = [.] x} in f <> x <> xs)
    X={repeat}
    M=@;
  |~> let {xs = @ x} in f <> x <> xs;
  -@-rules-3-lr
    ctx = (let G in let {xs = [.]} in f <> x <> xs)
    R=([.] x);
  <~> let {xs = {x}d^@} in f <> x <> xs;
  -@-rules-2
    ctx=(let G in [.])
    G=let {}
    x=xs
    X={x}
    N=(f <> x <> xs)
    M=(x:xs);
  |~> let {xs = x : xs} in f <> x <> xs;
} qed;

--inductive case (before induction)
proposition: G free(x f) |-
      let {xs = [0]s^(\x2. let {ys = repeat x2}
                           in x : ys) x} in f <> x <> xs
  |~> let { xs = let {g =[0,0]= (\a . \ as . a : as)}
                 in let {ys = repeat x} in g <> x <> ys}
      in f <> x <> xs;
proof: -simple -single{
  -spike-algebra-zero-stack-spike-lr
    ctx=(let G in let {xs= [.] x} in f <> x <> xs)
    M=(\x2. let {ys = repeat x2} in x : ys);
  <~> let {xs = (\x2. let {ys = repeat x2} in x : ys) x}
      in f <> x <> xs;
  -reduction-lr
    ctx=(let G in let {xs = [.]} in f <> x <> xs)
    w=1
    R=([.] x)
    V=(\x2. let {ys = repeat x2} in x : ys)
    X={}
    N=(let {ys' = repeat x} in x : ys');
  <~> let {xs = s^{}d^(let {ys = repeat x} in x : ys)}
      in f <> x <> xs;
  -dummy-ref-algebra-5-lr
    ctx=(let G in let {xs = s^[.]} in f <> x <> xs)
    M=(let {ys = repeat x} in x : ys);
  <~> let {xs = s^(let {ys = repeat x} in x : ys)} in f <> x <> xs;
  -spike-algebra-13
    ctx=(let G in let {xs = [.]} in f <> x <> xs)
    w=1
    M=(let {ys = repeat x} in x : ys);
  |~> let {xs = let {ys = repeat x} in x : ys} in f <> x <> xs;
  -- Start of the extra work that is needed because meta-variable M
  -- is not implemented.
  -dummy-ref-algebra-5-rl
    ctx=(let G in let {xs = let {ys = repeat x} in [.]}
               in f <> x <> xs)
    M=(x : ys);
  <~> let {xs = let {ys = repeat x} in {}d^(x : ys)}
      in f <> x <> xs;
  -spike-algebra-zero-stack-spike-rl
```

```
    ctx=(let G in let {xs = let {ys = repeat x} in [.]}
                  in f <> x <> xs)
    M=({}d^(x : ys));
  <~> let {xs = let {ys = repeat x} in [0]s^{}d^(x : ys)}
      in f <> x <> xs;
  -reduction-rl
    ctx=(let G in let {xs = let {ys = repeat x} in [.]}
                  in f <> x <> xs)
    N=(x : ys)
    w=0
    X={}
    R=([.] ys)
    V=(\as . x : as);
  <~> let {xs = let {ys = repeat x} in (\as . x : as) <> ys}
                in f <> x <> xs;
  -dummy-ref-algebra-5-rl
    ctx = (let G in let {xs = let {ys = repeat x}
                              in [.] <> ys} in f <> x <> xs)
    M=(\as . x : as);
  <~> let {xs = let {ys = repeat x}
                in {}d^(\as . x : as) <> ys}
      in f <> x <> xs;
  -spike-algebra-zero-stack-spike-rl
    ctx= (let G in let {xs = let {ys = repeat x}
                             in [.] <> ys} in f <> x <> xs)
    M=({}d^(\as . x : as));
  <~> let {xs = let {ys = repeat x} in [0]s^{}d^(\as . x : as) <> ys}
      in f <> x <> xs;
  -reduction-rl
    ctx= (let G in let {xs = let {ys = repeat x}
                             in [.] <> ys} in f <> x <> xs)
    w=0
    X={}
    N=(\as' . x : as')
    R=([.] x)
    V=(\a. \as . a : as);
  <~> let {xs = let {ys = repeat x} in (\a. \as . a : as) <> x <> ys}
                                  in f <> x <> xs;
  -spike-algebra-zero-heap-spike-rl
    ctx=(let G in let {xs = let {ys = repeat x}
                            in [.] <> x <> ys}
                  in f <> x <> xs)
    M=(\a. \as . a : as);
  <~> let {xs = let {ys = repeat x}
                in [0]h^(\a. \as . a : as) <> x <> ys}
      in f <> x <> xs;
```

```
-let-elim-rl
  ctx=(let G in let {xs = let {ys = repeat x}
                         in [.] <> x <> ys}
              in f <> x <> xs)
  M=(\a. \as . a : as)
  x=g
  v=0
  w=0;
<~> let {xs = let {ys = repeat x}
              in (let {g =[0,0]= (\a. \as . a : as)}
                  in g) <> x <> ys}
    in f <> x <> xs;
-let-R-rl
  ctx=(let G in let {xs = let {ys = repeat x}
                      in [.] <> ys}
             in f <> x <> xs)
  G=let {g =[0,0]= (\a. \as . a : as)}
  M=g
  R=([.] x)
  w=0;
<~> let {xs = let {ys = repeat x}
              in (let {g =[0,0]= (\a. \as . a : as)}
                  in g <> x) <> ys}
    in f <> x <> xs;
-let-R-rl
  ctx = (let G in let {xs = let {ys = repeat x}
                            in [.]}
                  in f <> x <> xs)
  G=let {g =[0,0]= (\a. \as . a : as)}
  M=(g <> x)
  R=([.] ys)
  w=0;
<~> let {xs = let {ys = repeat x}
              in let {g =[0,0]= (\a. \as . a : as)}
                 in g <> x <> ys}
    in f <> x <> xs;
-let-flatten-lr
  ctx = (let G in let {xs = [.]} in f <> x <> xs)
  G1=let {ys = repeat x}
  G2=let {g =[0,0]= (\a. \as . a : as)}
  M=(g <> x <> ys);
<~> let {xs = let { ys = repeat x
                  , g =[0,0]= (\a. \as . a : as)}
              in g <> x <> ys}
    in f <> x <> xs;
-let-flatten-rl
```

```
    ctx = (let G in let {xs = [.]} in f <> x <> xs)
    G1=let {g =[0,0]= (\a. \as . a : as)}
    G2=let {ys = repeat x}
    M=(g <> x <> ys);
  <~> let {xs = let {g =[0,0]= (\a. \as . a : as)}
                in let { ys = repeat x} in g <> x <> ys}
      in f <> x <> xs;
}
qed;
{-
This would be the induction step if induction was implemented,
but now it is in comments and there are two separate proofs instead.

  <~> let {xs = let {g =[0,0]= (\a. \as . a : as)}
                in let { ys = repeat^n x} in g <> x <> ys}
      in f <> x <> xs;

-ih
  ctx = [.]
  D = (let {xs = let {g =[0,0]= (\a. \as . a : as)}
                 in [.]}
       in f <> x <> xs)
  f=g
  x=x
  |~> let {xs = let {g =[0,0]= (\a. \as . a : as)}
                in let {ys = x : ys} in g <> x <> ys}
      in f <> x <> xs;
-}

-- inductive case (after induction)
proposition: G free(x f) |-
      let {xs = let {g =[0,0]= (\a. \as . a : as)}
                in let {ys = x : ys} in g <> x <> ys}
      in f <> x <> xs
      |~> let {xs = x : xs} in f <> x <> xs;
proof: -simple -single{
  -unfold-5-lr
    ctx=(let G in let {xs = [.]} in f <> x <> xs)
    G= let {}
    x=g
    V=(\a. \as . a : as)
    C=(let {ys = x : ys} in [.] <> x <> ys);
  <~> let {xs = let {g =[0,0]= (\a. \as . a : as)}
                in let {ys = x : ys}
                   in (\b. \bs . b : bs) <> x <> ys}
      in f <> x <> xs;
```

```
-balloon-reduction-lr
  ctx= (let G in let {xs = let {g =[0,0]= (\a. \as . a : as)}
                in let {ys = x : ys} in [.] <> ys}
      in f <> x <> xs)
  x=b
  M=(\bs . b : bs)
  y=x;
<~> let {xs = let {g =[0,0]= (\a. \as . a : as)}
              in let {ys = x : ys} in (\bs . x : bs) <> ys}
    in f <> x <> xs;
-balloon-reduction-lr
  ctx=(let G in let {xs = let {g =[0,0]= (\a. \as . a : as)}
                          in let {ys = x : ys} in [.]}
                in f <> x <> xs)
  M=(x : bs)
  x=bs
  y=ys;
<~> let {xs = let {g =[0,0]= (\a. \as . a : as)}
              in let {ys = x : ys} in x : ys}
    in f <> x <> xs;
-gc-lr
  ctx=(let G in  let {xs = [.]}
                 in f <> x <> xs)
  G1 = let {}
  G2 = let {g =[0,0]= (\a. \as . a : as)}
  X={}
  M=(let {ys = x : ys} in x : ys);
<~> let {xs = {}d^(let {}
                   in let {ys = x : ys} in x : ys)}
    in f <> x <> xs;
-dummy-ref-algebra-5-lr
  ctx=(let G in let {xs = [.]}
                in f <> x <> xs)
  M=(let {} in let {ys = x : ys} in x : ys);
<~> let {xs = (let {} in let {ys = x : ys} in x : ys)}
    in f <> x <> xs;
-empty-let-lr
  ctx = (let G in let {xs = [.]}
                  in f <> x <> xs)
  N=(let {ys = x : ys} in x : ys);
<~> let {xs = let {ys = x : ys} in x : ys}
    in f <> x <> xs;
-- End of the extra work that is needed because general
-- metavariable M is not implemented.
-value-merge'
  ctx=(let G in [.])
```

```
    G=let {}
    x=xs
    y=ys
    V=(x:ys)
    M=(f <> x <> xs);
  |~> let {xs = x : xs} in f <> x <> xs;
} qed;
```

### B.2.2 Case study 2: intermediate data structures

We substituted $M$ for $f\,any$ for similar reasons as in case study 1. For parsing reasons, we cannot use `(||)` to be the binary or-operator. Instead, we will use the operator `or` for this purpose. We acknowledge that this may be confusing, since `or` is the list operator for `or` in Haskell. However, the list operator will not be used in the proof, since

```
    any' p = or . map p
    or = foldr (||) False
```

is inlined to

```
    any' p = (foldr (||) False) . map p
```

and then, the `(||)` is changed to `or`, to create

```
    any' p = (foldr or False) . map p
```

This inlining is safe for space, since the definition of `or` is top-level, and can thus be set to have stack and heap weight 0 within weak space equivalence, so that `-unfold-5` can be applied.

#### B.2.2.1 Adding gadgets to any

```
bindings {
G = { or = \a. \b. case a of
                     { True -> True
                     , False -> b
                     }
    };
}


-- Add gadgets
proposition: G free(f)|-
  let {any = \p. \xs. case xs of
            { [] -> False
            , y:ys -> let { py = p y
                          , anypys = any p y ys}
                      in or py anypys}}
  in f any
   <~~>
  let {any = \p. \xs. [2]h^(case xs of
              { [] -> s^False
              , y:ys -> s^(let { z = @
                                , py = p <> y
                                , anypys = {z}d^(any <> p <> y <> ys)}
                            in or <> py <> anypys)})
  } in f any;
proof: -simple -single{
  -heap-spike-intro-lr
    ctx=(let G in (let {any = \p. \xs. [.] } in f any))
    M= (case xs of
              { [] -> False
              , y:ys -> let { py = p y
                             , anypys = any p y ys}
                         in or py anypys})
    w=2;
  <~~> let {any = \p. \xs. [2]h^(case xs of
                      { [] -> False
                      , y:ys -> let { py = p y
                                     , anypys = any p y ys}
                                 in or py anypys})
      } in f any;
  -stack-spike-intro-lr
    ctx = (let G in let {any = \p. \xs. [2]h^(case xs of
                      { [] -> False
                      , y:ys -> [.]}) } in f any)
    M=(let { py = p y
                , anypys = any p y ys}
              in or py anypys)
    w=1;
  <~~> let {any = \p. \xs. [2]h^(case xs of
                        { [] -> False
                        , y:ys -> s^(let { py = p y
                                          , anypys = any p y ys}
                                      in or py anypys)})
      } in f any;
  -stack-spike-intro-lr
    ctx = (let G in let {any = \p. \xs. [2]h^(case xs of
                          { [] -> [.]
                          , y:ys -> s^(let { py = p y
                                            , anypys = any p y ys}
                                        in or py anypys)})
                    } in f any)
    M=False
   w=1;
 <~~> let {any = \p. \xs. [2]h^(case xs of
                      { [] -> s^False
                      , y:ys -> s^(let { py = p y
                                       , anypys = any p y ys}
                               in or py anypys)})
      } in f any ;
 -balloon-intro-untyped-lr
   ctx = (let G in let {any = \p. \xs. [2]h^(case xs of
                        { [] -> s^False
                        , y:ys -> s^(let { py = [.]
                                         , anypys = any p y ys}
                                    in or py anypys)})
                 } in f any)
   M=p x=y;
 <~~> let {any = \p. \xs. [2]h^(case xs of
                      { [] -> s^False
                      , y:ys -> s^(let { py = p <> y
                                       , anypys = any p y ys}
                                  in or py anypys)}) } in f any;
 -balloon-intro-untyped-lr
   ctx=(let G in let {any = \p. \xs. [2]h^(case xs of
                        { [] -> s^False
                        , y:ys -> s^(let { py = p <> y
                                         , anypys = [.] y ys}
                                    in or py anypys)}) } in f any)
   M=any x=p;
 <~~> let {any = \p. \xs. [2]h^(case xs of
                      { [] -> s^False
                      , y:ys -> s^(let { py = p <> y
                                       , anypys = any <> p y ys}
                                  in or py anypys)}) } in f any;
 -balloon-intro-untyped-lr
   ctx=(let G in let {any = \p. \xs. [2]h^(case xs of
                        { [] -> s^False
                        , y:ys -> s^(let { py = p <> y
                                         , anypys = [.] ys}
                                    in or py anypys)}) } in f any)
   M=(any <> p) x=y;
 <~~> let {any = \p. \xs. [2]h^(case xs of
                      { [] -> s^False
                      , y:ys -> s^(let { py = p <> y
                                       , anypys = any <> p <> y ys}
                                  in or py anypys)}) } in f any;
 -balloon-intro-untyped-lr
   ctx=(let G in let {any = \p. \xs. [2]h^(case xs of
                         { [] -> s^False
                         , y:ys -> s^(let { py = p <> y
                                          , anypys = [.]}
                                      in or py anypys)}) } in f any)
  M=(any <> p <> y) x=ys;
<~~> let {any = \p. \xs. [2]h^(case xs of
                { [] -> s^False
                , y:ys -> s^(let { py = p <> y
                                 , anypys = any <> p <> y <> ys}
                             in or py anypys)}) } in f any;
-let-flatten-rl
  ctx=(let G in let {any = \p. \xs. [2]h^(case xs of
                               { [] -> s^False
                               , y:ys -> s^([.])}) } in f any)
  G1=let {anypys = any <> p <> y <> ys}
  G2=let {py = p<> y}
  M=(or py anypys);
<~> let {any = \p. \xs. [2]h^(case xs of
               { [] -> s^False
               , y:ys -> s^(let {anypys = any <> p <> y <> ys}
                            in let {py = p <> y}
                               in or py anypys)}) } in f any;
-dummy-bind-intro-lr
  ctx=(let G in let {any = \p. \xs. [2]h^(case xs of
                               { [] -> s^False
                               , y:ys -> s^([.])}) } in f any)
  x=anypys
  M=(any <> p <> y <> ys)
  N=(let {py = p <> y}
     in or py anypys)
  z=z;
<~~> let {any = \p. \xs. [2]h^(case xs of
           { [] -> s^False
           , y:ys -> s^(let { z=@
                            , anypys = {z}d^(any <> p <> y <> ys)}
                        in let {py = p <> y}
                           in or py anypys)}) } in f any;
-let-flatten-lr
  ctx= (let G in let {any = \p. \xs. [2]h^(case xs of
                         { [] -> s^False
                         , y:ys -> s^([.])}) } in f any)
  G1=let { z=@, anypys = {z}d^(any <> p <> y <> ys)}
  G2=let {py = p <> y}
  M=(or py anypys);
<~>  let {any = \p. \xs. [2]h^(case xs of
           { [] -> s^False
```

```
              , y:ys -> s^(let { z=@
                              , anypys = {z}d^(any <> p <> y <> ys)
                              , py = p <> y}
                           in or py anypys)}) } in f any;
  -balloon-intro-untyped-lr
    ctx = (let G
           in let {any = \p. \xs. [2]h^(case xs of
              { [] -> s^False
              , y:ys -> s^(let { z=@
                              , py = p <> y
                              , anypys = {z}d^(any <> p <> y <> ys)}
                           in [.] anypys)}) } in f any)
    M=or x=py;
  <~~> let {any = \p. \xs. [2]h^(case xs of
              { [] -> s^False
              , y:ys -> s^(let { z=@
                              , py = p <> y
                              , anypys = {z}d^(any <> p <> y <> ys)}
                           in or <> py anypys)}) } in f any;
  -balloon-intro-untyped-lr
    ctx = (let G in let {any = \p. \xs. [2]h^(case xs of
              { [] -> s^False
              , y:ys -> s^(let { z=@
                              , py = p <> y
                              , anypys = {z}d^(any <> p <> y <> ys)}
                           in [.])}) } in f any)
    M=(or <> py) x=anypys;
  <~~> let {any = \p. \xs. [2]h^(case xs of
              { [] -> s^False
              , y:ys -> s^(let { z=@
                              , py = p <> y
                              , anypys = {z}d^(any <> p <> y <> ys)}
                           in or <> py <> anypys)}) } in f any;
  }
  qed;
```

### B.2.2.2 Inductive case

```
bindings {
G = { any_a =[0,0]= \p1. \xs1. h^([2]case xs1 of
                  { [] -> s^False
                  , y1:ys1 -> s^(let { z1 = @
                                   , a1 = p1 <> y1
                                   , b1 = {z1}d^(any_a <> p1 <> ys1)}
                                in or <> a1 <> b1)})
    , foldr_a =[0,0]= \f2 . \ z2 . \l2 .
                  case l2 of
```

```
                    { [] -> z2
                    , a2:as2 -> let {t2 = foldr_a <> f2 <> z2 <> as2}
                                in f2 <> a2 <> t2}
    , map_a =[0,0]= \f3 . \l3 . case l3 of
        { [] -> []
        , a3:as3 -> let { h3 = f3 <> a3
                        , t3 = map_a <> f3 <> as3}
                    in h3:t3
        }
    , or =[0,0]= \a4. \b4. case a4 of
                         { True -> True
                         , False -> b4
                         }
    , false =[0,0]= False
    };
}

-- pre-induction
proposition: G free(p xs) |-
  let { b = map_a <> p <> xs}
  in foldr_a <> or <> false <> b
  <~>
  h^([2] case xs of
    { [] -> s^false
    , b : bs -> s^(let { z = @
                       , ds = {z}d^(let {cs = map_a <> p <> bs}
                                    in foldr_a <> or <> false <> cs)}
                   in let {c = p <> b}
                      in or <> c <> ds)});

proof: -simple -single {
  let {b = map_a <> p <> xs}
  in foldr_a <> or <> false <> b;
  -unfold-5-lr
    ctx = [.]
    G= let { any_a =[0,0]= \p1. \xs1. h^([2]case xs1 of
                   { [] -> s^False
                   , y1:ys1 -> s^(let { z1 = @
                                      , a1 = p1 <> y1
                                      , b1 = {z1}d^(any_a <> p1 <> ys1)}
                                  in or <> a1 <> b1)})
           , map_a =[0,0]= \f3 . \l3 . case l3 of
               { [] -> []
               , a3:as3 -> let { h3 = f3 <> a3
                               , t3 = map_a <> f3 <> as3}
                           in h3:t3
```

```
              }
          , or =[0,0]= \a4. \b4. case a4 of
                                { True -> True
                                , False -> b4
                                }
          , false =[0,0]= False
          }
   x = foldr_a
   V = (\f5 . \ z5 . \l5 .
          case l5 of
            { [] -> z5
            , a5:as5 -> let {t5 = foldr_a <> f5 <> z5 <> as5}
                        in f5 <> a5 <> t5})
   C = (let { b = map_a <> p <> xs}
        in [.] <> or <> false <> b);
 <~>
 let { b = map_a <> p <> xs}
 in (\f5 . \ z5 . \l5 .
        case l5 of
          { [] -> z5
          , a5:as5 -> let {t5 = foldr_a <> f5 <> z5 <> as5}
                      in f5 <> a5 <> t5}) <> or <> false <> b;
 -balloon-reduction-lr
   ctx = (let G in let { b = map_a <> p <> xs}
                   in [.] <> false <> b)
   M = (\ z5 . \l5 .
          case l5 of
            { [] -> z5
            , a5:as5 -> let {t5 = foldr_a <> f5 <> z5 <> as5}
                        in f5 <> a5 <> t5})
   x=f5 y=or;
 <~>
 let { b = map_a <> p <> xs}
 in (\ z5 . \l5 .
        case l5 of
          { [] -> z5
          , a5:as5 -> let {t5 = foldr_a <> or <> z5 <> as5}
                      in or <> a5 <> t5}) <> false <> b;
 -balloon-reduction-lr
   ctx = (let G in let {b = map_a <> p <> xs}
                   in [.] <> b)
   x=z5 y=false
   M=(\l5 . case l5 of
            { [] -> z5
            , a5:as5 -> let {t5 = foldr_a <> or <> z5 <> as5}
                        in or <> a5 <> t5});
```

```
<~>
let { b = map_a <> p <> xs}
in (\l5 . case l5 of
         { [] -> false
         , a5:as5 -> let {t5 = foldr_a <> or <> false <> as5}
                     in or <> a5 <> t5}) <> b;
-balloon-reduction-lr
  ctx = (let G in let { b = map_a <> p <> xs}
                  in [.])
  x=l5 y = b
  M=(case l5 of
           { [] -> false
           , a5:as5 -> let {t5 = foldr_a <> or <> false <> as5}
                       in or <> a5 <> t5});
<~>
let { ys5 = map_a <> p <> xs}
in (case ys5 of
         { [] -> false
         , a5:as5 -> let {t5 = foldr_a <> or <> false <> as5}
                     in or <> a5 <> t5});
-let-R-lr
  ctx = (let G in [.])
  G = let {ys5 = map_a <> p <> xs}
  w=1
  R=(case [.] of { [] -> false
                 , a5:as5 ->
                     let {t5 = foldr_a <> or <> false <> as5}
                     in or <> a5 <> t5})
  M=ys5;
<~> (case (let {ys5 = map_a <> p <> xs} in ys5) of
                { [] -> false
                , a5:as5 ->
                    let {t5 = foldr_a <> or <> false <> as5}
                    in or <> a5 <> t5});
-let-elim-lr
  x=ys5
  v=1
  w=1
  M=(map_a <> p <> xs)
  ctx=(let G in (case [.] of
                 { [] -> false
                 , a5:as5 ->
                     let {t5 = foldr_a <> or <> false <> as5}
                     in or <> a5 <> t5}));
<~> (case h^(map_a <> p <> xs) of
                { [] -> false
```

```
                  , a5:as5 ->
                      let {t5 = foldr_a <> or <> false <> as5}
                      in or <> a5 <> t5});
-spike-algebra-2-lr
  ctx=(let G in [.])
  w=1
  R=(case [.] of
              { [] -> false
              , a5:as5 -> let {t5 = foldr_a <> or <> false <> as5}
                                in or <> a5 <> t5})
  v=1
  M=(map_a <> p <> xs);
<~> h^(case map_a <> p <> xs of
                { [] -> false
                , a5:as5 ->
                    let {t5 = foldr_a <> or <> false <> as5}
                    in or <> a5 <> t5});
-unfold-5-lr
  ctx= [.]
  G= let { any_a =[0,0]= \p1. \xs1. h^([2]case xs1 of
                { [] -> s^False
                , y1:ys1 -> s^(let { z1 = @
                                    , a1 = p1 <> y1
                                    , b1 = {z1}d^(any_a <> p1 <> ys1)}
                                in or <> a1 <> b1)})
         , foldr_a =[0,0]= \f2 . \ z2 . \l2 .
                case l2 of
                  { [] -> z2
                  , a2:as2 -> let {t2 = foldr_a <> f2 <> z2 <> as2}
                              in f2 <> a2 <> t2}
         , or =[0,0]= \a4. \b4. case a4 of
                              { True -> True
                              , False -> b4
                              }
         , false =[0,0]= False
         }
  V= (\f3 . \l3 . case l3 of
      { [] -> []
      , a3:as3 -> let { h3 = f3 <> a3
                      , t3 = map_a <> f3 <> as3}
                  in h3:t3})
  x=map_a
  C=(h^(case [.] <> p <> xs of
        { [] -> false
        , a5:as5 -> let {t5 = foldr_a <> or <> false <> as5}
                    in or <> a5 <> t5}));
```

```
<~> h^(case (\f7 . \l7 . case l7 of
                        { [] -> []
                        , a7:as7 ->
                            let { h7 = f7 <> a7
                                , t7 = map_a <> f7 <> as7}
                            in h7:t7}) <> p <> xs of
                { [] -> false
                , a5:as5 ->
                    let {t5 = foldr_a <> or <> false <> as5}
                    in or <> a5 <> t5});
-balloon-reduction-lr
  ctx=(let G in h^(case [.] <> xs of
                  { [] -> false
                  , a5:as5 ->
                      let {t5 = foldr_a <> or <> false <> as5}
                      in or <> a5 <> t5}))
  x=f7 y=p
  M=(\l8 . case l8 of
           { [] -> []
           , a8:as8 -> let { h8 = f7 <> a8
                           , t8 = map_a <> f7 <> as8}
                       in h8:t8});
<~> h^(case (\l7 . case l7 of
                      { [] -> []
                      , a7:as7 -> let { h7 = p <> a7
                                      , t7 = map_a <> p <> as7}
                                  in h7:t7}) <> xs of
                { [] -> false
                , a5:as5 ->
                    let {t5 = foldr_a <> or <> false <> as5}
                    in or <> a5 <> t5});
-balloon-reduction-lr
  ctx=(let G in h^(case [.] of
                  { [] -> false
                  , a5:as5 ->
                      let {t5 = foldr_a <> or <> false <> as5}
                      in or <> a5 <> t5}))
  x=l7 y=xs
  M=(case l7 of
       { [] -> []
       , a7:as7 -> let { h7 = p <> a7
                       , t7 = map_a <> p <> as7}
                   in h7:t7});
<~> h^(case case xs of
                      { [] -> []
                      , a7:as7 -> let { h7 = p <> a7
```

```
                                        , t7 = map_a <> p <> as7}
                                    in h7:t7} of
                  { [] -> false
                  , a5:as5 ->
                      let {t5 = foldr_a <> or <> false <> as5}
                      in or <> a5 <> t5});
  -R-case-lr
    ctx= (let G in h^[.])
    R=(case [.] of
                { [] -> false
                , a5:as5 -> let {t5 = foldr_a <> or <> false <> as5}
                               in or <> a5 <> t5})
    w=1
    v=1
    M=xs
    pat_i=patterns [ [], a10:as10 ]
    N_i=terms [ [],  let { h7 = p <> a10
                         , t7 = map_a <> p <> as10}
                     in h7:t7];
  <~> h^ ([2] case xs of
        { [] -> case [] of
                  { [] -> false
                  , a10:as10 ->
                      let {t5 = foldr_a <> or <> false <> as10}
                      in or <> a10 <> t5}
        , b:bs -> case let { h7 = p <> b
                           , t7 = map_a <> p <> bs}
                       in h7:t7 of
                   { [] -> false
                   , a9:as9 ->
                       let {t9 = foldr_a <> or <> false <> as9}
                       in or <> a9 <> t9}
        });
  -reduction-lr
    ctx = (let G in h^ ([2] case xs of
           { [] -> [.]
           , b:bs -> case let { h7 = p <> b
                              , t7 = map_a <> p <> bs}
                          in h7:t7 of
                      { [] -> false
                      , a9:as9 ->
                          let {t9 = foldr_a <> or <> false <> as9}
                          in or <> a9 <> t9}
           }))
    w=1
    R=(case [.] of
                { [] -> false
                , a10:as10 -> let {t5 = foldr_a <> or <> false <> as10}
                               in or <> a10 <> t5})
    V= []
    X={or foldr_a}
    N=false;
  <~> h^ ([2] case xs of
        { [] -> s^{or foldr_a}d^false
        , b:bs -> case let { h7 = p <> b
                           , t7 = map_a <> p <> bs}
                       in h7:t7 of
                   { [] -> false
                   , a9:as9 ->
                       let {t9 = foldr_a <> or <> false <> as9}
                       in or <> a9 <> t9}
        });
  -dummy-ref-algebra-7-rl
    ctx=(let G in h^ ([2] case xs of
           { [] -> s^[.]
           , b:bs -> case let { h7 = p <> b
                              , t7 = map_a <> p <> bs}
                          in h7:t7 of
                      { [] -> false
                      , a9:as9 ->
                          let {t9 = foldr_a <> or <> false <> as9}
                          in or <> a9 <> t9}
           }))
    X={or}
    Y={foldr_a}
    M=false;
  <~> h^ ([2] case xs of
        { [] -> s^{or}d^{foldr_a}d^false
        , b:bs -> case let { h7 = p <> b
                           , t7 = map_a <> p <> bs}
                       in h7:t7 of
                   { [] -> false
                   , a9:as9 ->
                       let {t9 = foldr_a <> or <> false <> as9}
                       in or <> a9 <> t9}
        });
  -dummy-ref-algebra-3-lr
    ctx = [.]
    G= let { any_a =[0,0]= \p1. \xs1. h^([2]case xs1 of
                 { [] -> s^False
                 , y1:ys1 -> s^(let { z1 = @
                                    , a1 = p1 <> y1
```

```
                              , b1 = {z1}d^(any_a <> p1 <> ys1)}
                          in or <> a1 <> b1)})
        , map_a =[0,0]= \f3 . \l3 . case l3 of
            { [] -> []
            , a3:as3 -> let { h3 = f3 <> a3
                            , t3 = map_a <> f3 <> as3}
                        in h3:t3
            }
        , or =[0,0]= \a4. \b4. case a4 of
                                { True -> True
                                , False -> b4
                                }
        , false =[0,0]= False
        }
    x=foldr_a
    w=0
    V= (\f2 . \ z2 . \l2 .
                            case l2 of
                              { [] -> z2
                              , a2:as2 ->
                                  let {t2 = foldr_a <> f2 <> z2 <> as2}
                                  in f2 <> a2 <> t2})
    C=( h^ ([2] case xs of
          { [] -> s^{or}d^[.]
          , b:bs -> case let { h7 = p <> b
                                , t7 = map_a <> p <> bs}
                            in h7:t7 of
                         { [] -> false
                         , a9:as9 ->
                             let {t9 = foldr_a <> or <> false <> as9}
                             in or <> a9 <> t9}
          }))
    M=false;
  <~> h^ ([2] case xs of
        { [] -> s^{or}d^{}d^false
        , b:bs -> case let { h7 = p <> b
                              , t7 = map_a <> p <> bs}
                          in h7:t7 of
                       { [] -> false
                       , a9:as9 ->
                           let {t9 = foldr_a <> or <> false <> as9}
                           in or <> a9 <> t9}
        });
  -dummy-ref-algebra-5-lr
  ctx=(let G in h^ ([2] case xs of
        { [] -> s^{or}d^[.]
```

```
      , b:bs -> case let { h7 = p <> b
                         , t7 = map_a <> p <> bs}
                     in h7:t7 of
                  { [] -> false
                  , a9:as9 ->
                      let {t9 = foldr_a <> or <> false <> as9}
                      in or <> a9 <> t9}
      }))
  M=false;
<~> h^ ([2] case xs of
      { [] -> s^{or}d^false
      , b:bs -> case let { h7 = p <> b
                         , t7 = map_a <> p <> bs}
                     in h7:t7 of
                  { [] -> false
                  , a9:as9 ->
                      let {t9 = foldr_a <> or <> false <> as9}
                      in or <> a9 <> t9}
      });
-dummy-ref-algebra-3-lr
  ctx= [.]
  G= let { any_a =[0,0]= \p1. \xs1. h^([2]case xs1 of
             { [] -> s^False
             , y1:ys1 -> s^(let { z1 = @
                                , a1 = p1 <> y1
                                , b1 = {z1}d^(any_a <> p1 <> ys1)}
                            in or <> a1 <> b1)})
      , foldr_a =[0,0]= \f2 . \ z2 . \l2 .
          case l2 of
            { [] -> z2
            , a2:as2 -> let {t2 = foldr_a <> f2 <> z2 <> as2}
                        in f2 <> a2 <> t2}
      , map_a =[0,0]= \f3 . \l3 . case l3 of
          { [] -> []
          , a3:as3 -> let { h3 = f3 <> a3
                          , t3 = map_a <> f3 <> as3}
                      in h3:t3
          }
      , false =[0,0]= False
      }
  x=or
  w=0
  V=(\a4. \b4. case a4 of
                { True -> True
                , False -> b4
                })
```

```
    M=false
    C=(h^ ([2] case xs of
          { [] -> s^[.]
          , b:bs -> case let { h7 = p <> b
                                , t7 = map_a <> p <> bs}
                            in h7:t7 of
                         { [] -> false
                         , a9:as9 ->
                             let {t9 = foldr_a <> or <> false <> as9}
                             in or <> a9 <> t9}
          }));
  <~> h^ ([2] case xs of
        { [] -> s^{}d^false
        , b:bs -> case let { h7 = p <> b
                              , t7 = map_a <> p <> bs}
                          in h7:t7 of
                       { [] -> false
                       , a9:as9 ->
                           let {t9 = foldr_a <> or <> false <> as9}
                           in or <> a9 <> t9}
        });
  -dummy-ref-algebra-5-lr
    ctx=(let G in h^ ([2] case xs of
          { [] -> s^[.]
          , b:bs -> case let { h7 = p <> b
                                , t7 = map_a <> p <> bs}
                            in h7:t7 of
                         { [] -> false
                         , a9:as9 ->
                             let {t9 = foldr_a <> or <> false <> as9}
                             in or <> a9 <> t9}
          }))
    M=false;
  <~> h^ ([2] case xs of
        { [] -> s^false
        , b:bs -> case let { h7 = p <> b
                              , t7 = map_a <> p <> bs}
                          in h7:t7 of
                       { [] -> false
                       , a9:as9 ->
                           let {t9 = foldr_a <> or <> false <> as9}
                           in or <> a9 <> t9}
        });
  -let-R-rl
    ctx=(let G
         in h^ ([2] case xs of
          { [] -> s^false
          , b:bs -> [.]
          }))
    R=(case [.] of
                { [] -> false
                , a9:as9 ->
                       let {t9 = foldr_a <> or <> false <> as9}
                       in or <> a9 <> t9})
    G=let { h7 = p <> b, t7 = map_a <> p <> bs}
    M=(h7:t7)
    w=1;
  <~> h^ ([2] case xs of
        { [] -> s^false
        , b:bs -> let { h7 = p <> b
                         , t7 = map_a <> p <> bs}
                     in case h7:t7 of
                       { [] -> false
                       , a9:as9 ->
                           let {t9 = foldr_a <> or <> false <> as9}
                           in or <> a9 <> t9}
        });
  -reduction-lr
    ctx=(let G
         in h^ ([2] case xs of
          { [] -> s^false
          , b:bs -> let { h7 = p <> b
                           , t7 = map_a <> p <> bs}
                       in [.]
          }))
    w=1
    R=(case [.] of
      { [] -> false
      , a9:as9 ->
          let {t9 = foldr_a <> or <> false <> as9}
          in or <> a9 <> t9})
    V=(h7:t7)
    X={}
    N=(let {t = foldr_a <> or <> false <> t7}
       in or <> h7 <> t);
  <~> h^ ([2] case xs of
        { [] -> s^false
        , b:bs -> let { h7 = p <> b
                         , t7 = map_a <> p <> bs}
                     in s^{}d^(let {t9 = foldr_a <> or <> false <> t7}
                               in or <> h7 <> t9)
        });
```

```
-dummy-ref-algebra-5-lr
  ctx=(let G
       in h^ ([2] case xs of
        { [] -> s^false
        , b:bs -> let { h7 = p <> b
                      , t7 = map_a <> p <> bs}
                  in s^[.]
        }))
  M=(let {t9 = foldr_a <> or <> false <> t7}
            in or <> h7 <> t9);
<~>  h^ ([2] case xs of
      { [] -> s^false
      , b:bs -> let { h7 = p <> b
                    , t7 = map_a <> p <> bs}
                in s^(let {t9 = foldr_a <> or <> false <> t7}
                      in or <> h7 <> t9)
      });
-spike-algebra-3-lr
  ctx=(let G
      in h^ ([2] case xs of
        { [] -> s^false
        , b:bs -> [.]
        }))
  G= let { h7 = p <> b
         , t7 = map_a <> p <> bs}
  v=1
  M=(let {t9 = foldr_a <> or <> false <> t7}
     in or <> h7 <> t9);
<~> h^ ([2] case xs of
      { [] -> s^false
      , b:bs -> s^(let { h7 = p <> b
                       , t7 = map_a <> p <> bs}
                   in let {t9 = foldr_a <> or <> false <> t7}
                      in or <> h7 <> t9)
      });
-let-flatten-lr
  ctx = (let G in h^ ([2] case xs of
                  { [] -> s^false
                  , b:bs -> s^[.]
                  }))
  G1 = let { h7 = p <> b, t7 = map_a <> p <> bs}
  G2= let {t9 = foldr_a <> or <> false <> t7}
  M=(or <> h7 <> t9);
<~> h^ ([2] case xs of
      { [] -> s^false
      , b:bs -> s^(let { c = p <> b
                        , cs = map_a <> p <> bs
                        , ds = foldr_a <> or <> false <> cs}
                    in or <> c <> ds)
        });
  -let-flatten-rl
    ctx=(let G
        in h^ ([2] case xs of
          { [] -> s^false
          , b:bs -> s^[.]
          }))
    G1= let {cs = map_a <> p <> bs
            , ds = foldr_a <> or <> false <> cs}
    G2= let { c = p <> b}
    M=(or <> c <> ds);
  <~>  h^ ([2] case xs of
        { [] -> s^false
        , b:bs -> s^(let { cs = map_a <> p <> bs
                        , ds = foldr_a <> or <> false <> cs}
                    in let {c = p <> b}
                       in or <> c <> ds)
        });
  -let-let'-lr
    ctx=(let G in h^ ([2] case xs of
                    { [] -> s^false
                    , b:bs -> s^[.]
                    }))
    G1=let {cs = map_a <> p <> bs}
    x=ds
    v=1 w=1
    M=(foldr_a <> or <> false <> cs)
    N=(let {c = p <> b}
       in or <> c <> ds)
    G2=let {z = @};
  <~> h^ ([2] case xs of
        { [] -> s^false
        , b:bs ->
            s^(let { z = @
                   , ds = {z}d^(let {cs = map_a <> p <> bs}
                                in foldr_a <> or <> false <> cs)}
               in let {c = p <> b}
                  in or <> c <> ds)
        });
} qed;

{-
Induction is not implemented, so this is in comments. However, if it was
```

```
implemented, the step would probably look like this.
  -ih
    ctx= [.]
    G=G
    C=(h^ ([2] case xs of
          { [] -> s^false
          , b:bs -> s^(let { z = @
                           , ds = {z}d^([.])}
                       in let {c = p <> b}
                          in or <> c <> ds)
          }))
    p=p
    xs=bs
-}

-- post-induction
proposition: G free(p xs) |-
  h^ ([2] case xs of
            { [] -> s^false
            , b:bs -> s^(let { z = @
                             , ds = {z}d^(any_a <> p <> bs)}
                         in let {c = p <> b}
                            in or <> c <> ds)
            })
  <~>
  any_a <> p <> xs;
proof: -simple -single {
  h^ ([2] case xs of
            { [] -> s^false
            , b:bs -> s^(let { z = @
                             , ds = {z}d^(any_a <> p <> bs)}
                         in let {c = p <> b}
                            in or <> c <> ds)
            });
  -let-flatten-lr
    ctx=(let G in h^ ([2] case xs of
              { [] -> s^false
              , b:bs -> s^[.]
              }))
    G1=let { z = @, ds = {z}d^(any_a <> p <> bs)}
    G2=let {c = p <> b}
    M=(or <> c <> ds);
  <~> h^ ([2] case xs of
            { [] -> s^false
            , b:bs -> s^(let { z = @
                             , ds = {z}d^(any_a <> p <> bs)
                             , c = p <> b}
                         in or <> c <> ds)
           });
-unfold-5-lr
  ctx = [.]
  G=let { any_a =[0,0]= \p1. \xs1. h^([2]case xs1 of
             { [] -> s^False
             , y1:ys1 -> s^(let { z1 = @
                                , a1 = p1 <> y1
                                , b1 = {z1}d^(any_a <> p1 <> ys1)}
                           in or <> a1 <> b1)})
      , foldr_a =[0,0]= \f2 . \ z2 . \l2 .
           case l2 of
             { [] -> z2
             , a2:as2 -> let {t2 = foldr_a <> f2 <> z2 <> as2}
                          in f2 <> a2 <> t2}
      , map_a =[0,0]= \f3 . \l3 . case l3 of
           { [] -> []
           , a3:as3 -> let { h3 = f3 <> a3
                           , t3 = map_a <> f3 <> as3}
                       in h3:t3
           }
      , or =[0,0]= \a4. \b4. case a4 of
                             { True -> True
                             , False -> b4
                             }
      }
  x=false
  V=False
  C=(h^ ([2] case xs of
             { [] -> s^[.]
             , b:bs -> s^(let { z = @
                              , ds = {z}d^(any_a <> p <> bs)
                              , c = p <> b}
                          in or <> c <> ds)
             }));
<~> h^ ([2] case xs of
           { [] -> s^False
           , b:bs -> s^(let { z = @
                            , ds = {z}d^(any_a <> p <> bs)
                            , c = p <> b}
                        in or <> c <> ds)
           });
-spike-algebra-zero-stack-spike-rl
    ctx=(let G in [.])
    M=(h^ ([2] case xs of
                { [] -> s^False
                , b:bs -> s^(let { z = @
                                 , ds = {z}d^(any_a <> p <> bs)
                                 , c = p <> b}
                             in or <> c <> ds)
                }));
  <~> [0]s^h^ ([2] case xs of
            { [] -> s^False
            , b:bs -> s^(let { z = @
                             , ds = {z}d^(any_a <> p <> bs)
                             , c = p <> b}
                         in or <> c <> ds)
            });
    -dummy-ref-algebra-5-rl
      ctx=(let G in [0]s^[.])
      M=(h^ ([2] case xs of
                { [] -> s^False
                , b:bs -> s^(let { z = @
                                 , ds = {z}d^(any_a <> p <> bs)
                                 , c = p <> b}
                             in or <> c <> ds)
                }));
  <~> [0]s^{}d^h^ ([2] case xs of
            { [] -> s^False
            , b:bs -> s^(let { z = @
                             , ds = {z}d^(any_a <> p <> bs)
                             , c = p <> b}
                         in or <> c <> ds)
            });
  -reduction-rl
    ctx = (let G in [.])
    w=0
    R=([.] xs)
    V=(\xs2 . h^ ([2] case xs2 of
              { [] -> s^False
              , b:bs -> s^(let { z9 = @
                               , ds9 = {z9}d^(any_a <> p <> bs)
                               , c9 = p <> b}
                           in or <> c9 <> ds9)
              }))
    N=(h^ ([2] case xs of
              { [] -> s^False
              , b8:bs8 -> s^(let { z8 = @
                               , ds8 = {z8}d^(any_a <> p <> bs8)
                               , c8 = p <> b8}
                           in or <> c8 <> ds8)
```

```
          }))
  X={};
<~> (\xs2 . h^ ([2] case xs2 of
          { [] -> s^False
          , b:bs -> s^(let { z = @
                          , ds = {z}d^(any_a <> p <> bs)
                          , c = p <> b}
                        in or <> c <> ds)
          })) <> xs;
-spike-algebra-zero-stack-spike-rl
  ctx = (let G in [.] <> xs)
  M=(\xs2 . h^ ([2] case xs2 of
            { [] -> s^False
            , b:bs -> s^(let { z = @
                            , ds = {z}d^(any_a <> p <> bs)
                            , c = p <> b}
                          in or <> c <> ds)
            }));
<~> [0]s^(\xs2 . h^ ([2] case xs2 of
          { [] -> s^False
          , b:bs -> s^(let { z = @
                          , ds = {z}d^(any_a <> p <> bs)
                          , c = p <> b}
                        in or <> c <> ds)
          })) <> xs;
-dummy-ref-algebra-5-rl
  ctx=(let G in [0]s^[.] <> xs)
  M=(\xs2 . h^ ([2] case xs2 of
            { [] -> s^False
            , b:bs -> s^(let { z = @
                            , ds = {z}d^(any_a <> p <> bs)
                            , c = p <> b}
                          in or <> c <> ds)
            }));
<~> [0]s^{}d^(\xs2 . h^ ([2] case xs2 of
          { [] -> s^False
          , b:bs -> s^(let { z = @
                          , ds = {z}d^(any_a <> p <> bs)
                          , c = p <> b}
                        in or <> c <> ds)
          })) <> xs;
-reduction-rl
  ctx=(let G in [.] <> xs)
  w=0
  R=([.] p)
  X={}
```

```
    V=(\q . \xs3 . h^ ([2] case xs3 of
              { [] -> s^False
              , e:es -> s^(let { z' = @
                                 , fs = {z'}d^(any_a <> q <> es)
                                 , y = q <> e}
                             in or <> y <> fs)
              }))
    N=(\xs2 . h^ ([2] case xs2 of
              { [] -> s^False
              , b:bs -> s^(let { z = @
                                 , ds = {z}d^(any_a <> p <> bs)
                                 , c = p <> b}
                             in or <> c <> ds)
              }));
  <~> (\p9 . \xs9 . h^ ([2] case xs9 of
            { [] -> s^False
            , b:bs -> s^(let { z = @
                               , ds = {z}d^(any_a <> p9 <> bs)
                               , c = p9 <> b}
                           in or <> c <> ds)
            })) <> p <> xs;
  -unfold-5-rl
    ctx = [.]
    G=let {foldr_a =[0,0]= \f2 . \ z2 . \l2 .
                           case l2 of
                             { [] -> z2
                             , a2:as2 ->
                                 let {t2 = foldr_a <> f2 <> z2 <> as2}
                                 in f2 <> a2 <> t2}
        , map_a =[0,0]= \f3 . \l3 . case l3 of
            { [] -> []
            , a3:as3 -> let { h3 = f3 <> a3
                             , t3 = map_a <> f3 <> as3}
                          in h3:t3
            }
        , or =[0,0]= \a4. \b4. case a4 of
                              { True -> True
                              , False -> b4
                              }
        , false =[0,0]= False
        }
    x=any_a
    V=( \p1. \xs1. h^([2]case xs1 of
          { [] -> s^False
          , y1:ys1 -> s^(let { z1 = @
                             , a1 = p1 <> y1
```

```
                                    , b1 = {z1}d^(any_a <> p1 <> ys1)}
                                in or <> a1 <> b1)}))
    C=([.] <> p <> xs);
  <~> any_a <> p <> xs;
} qed;
```