Space Optimizations in Deterministic and Concurrent Call-by-Need Functional Programming Languages

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik
der Johann Wolfgang Goethe-Universität
in Frankfurt am Main

von
Nils Dallmeyer
aus Hanau

Frankfurt (2020)
(D 30)

vom Fachbereich Informatik und Mathematik der
Johann Wolfgang Goethe-Universität als Dissertation angenommen.

Dekan:	Prof. Dr. Lars Hedrich
Gutachter:	Prof. Dr. Manfred Schmidt-Schauß
	Prof. Dr. Nils Bertschinger
Datum der Disputation:	02.10.2020

Abstract

In this thesis the space consumption and runtime in lazy functional languages is investigated. Both a pure and a concurrent scenario is analyzed. We go through the chapters in detail.

Chapter 2

This chapter contains preliminaries. The calculi LRP (SSS16b, SS15, SS14, SSS15b) and CHF* (SSSD18) are recalled. LRP is a polymorphically typed lazy lambda calculus extended by recursive `let`-expressions, `seq`-expressions and `case`-expressions as in Haskell. CHF* uses an action layer to implement monadic features, that might introduce side-effects and a functional layer that can be compared to LRP. CHF* is a slightly modified variant of CHF (SS11, SSS12), we show the semantic equivalence of both calculi.

The abstract machines M1, IOM1 and CIOM1 are recalled (see (Sab12)). The abstract machine M1 is compatible to LRP and CIOM1 is compatible to CHF*.

We also sketch a frequently used proof technique using forking diagrams and give definitions of often used functions in LRP and CHF*.

Chapter 3

The calculus LRPgc is introduced, that is LRP extended by a classical implementable garbage collector that is applied after each normal order reduction step. The operational semantics is called *normal order*. The `size` of expressions is defined precisely and also the space measure *spmax* is given, that yields the overall space consumption of an LRPgc-program, that is the maximal size of only garbage-free expressions that occurred during the whole normal order reduction sequence.

We consider program transformations in form of source-to-source transformations, hence a transformation is applied on the syntactical layer and also the result remains on this layer. The notions of transformations to be a *space improvement* or *space equivalence* are introduced. The intuition of a space improvement is, that it can be applied on any possible subexpression, where the pattern of the transformation matches, without increasing the space consumption in all of these cases – for at least one case it must decrease and if this is not the case, then it is a space equivalence. These definitions make use of contexts. A context $\mathbb{C}$ is an expression with a single hole, where expressions can be put in, i.e. $\mathbb{C}[\cdot]$ where $[\cdot]$ is the hole. The strong requirement to apply space improvements and equivalences at any matching subexpression can be implemented using contexts:

Let s and t be LRPgc-expressions. s is a space improvement of t, if the inequation $\mathit{spmax}(\mathbb{C}[s]) \leq \mathit{spmax}(\mathbb{C}[t])$ holds for all contexts $\mathbb{C}$. The definition for space equivalences is analogous requiring $\mathit{spmax}(\mathbb{C}[s]) = \mathit{spmax}(\mathbb{C}[t])$.

The amount of contexts that need to be considered for a proof of the space improvement or space equivalence property of a certain transformation is very high. Therefore we show so-called context lemmas that allow to conclude, that if the space improvement or equivalence property holds for reduction contexts (i.e. contexts that model the call-by-need strategy used by normal order reduction), the property also holds for all contexts.

For each considered transformation T all overlaps between normal order reduction and T need to be considered. This is done using sets of forking diagrams and then induction is applied to show the corresponding space property. With the use of the context lemmas, the amount of forking diagrams can be reduced, easing the proofs. Following this approach we show the space improvement and space equivalence properties of several transformations.

Moreover we define a transformation to be a *space leak* if there is at least one case where the space increases uncontrollably. We show for certain transformations that they are space leaks: The copying of functions, the change of evaluation order using strictness knowledge and the elimination of common subexpressions.

Further space analyses are performed for the case where extra knowledge is present about the overall space usage.

We also introduce the abstract machine $\mathrm{M1}_{sp}$ for space measurement, that is an adaption of the abstract machine M1 and use an implementation of $\mathrm{M1}_{sp}$ for several more complex space and runtime analyses. We show the correctness of this machine w.r.t. measurements. The approach using the abstract machine also allows to try different configurations of the garbage collector, allowing us to analyze the impact of garbage collection.

This chapter is based on (SSD18, SSD17).

Chapter 4

The already mentioned space analyses for LRP depend on the used garbage collector. To keep the space improvement and space equivalences results more independent of garbage collection, the so-called *total garbage collection* is introduced.

As calculus LR is used, i.e. LRP without types. Total garbage collection replaces a subexpression by a non-terminating constant with size zero, if this does not affect the overall termination of the program. Such a garbage collector is not implementable, since the garbage collector needs to be able to decide the termination of a program, but it can be still used for space analyses.

We transfer the `size`-measure from LRP to LR and define the space-measure *sps* for evaluation sequences of LR as the maximal `size`, where only garbage-free sizes w.r.t. total garbage collection are considered. The notions of *total space improvement* and *total space equivalence* are introduced. As also for the space improvements and equivalences mentioned above, the idea is, that such a transformation can be applied to any subexpression of an LR-expression, if the pattern of the transformation matches, while preserving the space property:

Let s and t be LR-expressions. s is a total space improvement of t, if the inequation *sps*$(\mathbb{C}[s]) \leq$ *sps*$(\mathbb{C}[t])$ holds for all contexts $\mathbb{C}$. The definition for total space equivalences is analogous requiring *sps*$(\mathbb{C}[s]) =$ *sps*$(\mathbb{C}[t])$.

Also for total garbage collection the amount of contexts that need to be considered in the proofs of total space improvement or equivalence properties of transformations is too high. Therefore context lemmas are showed, that allow to perform the proof using forking diagrams under reduction contexts and then conclude that the property also holds if all contexts are considered. Following this approach we show the total space improvement and total space equivalence properties of several transformations.

The notion of space leak is transferred from LRP to LR and used to show the space leak property of certain transformations: The copying of functions, the change of evaluation order using strictness knowledge and the elimination of common subexpressions.

Total garbage collection is classified w.r.t. its optimality. Since the approach of total garbage collection replaces subexpressions by non-terminating constants with size zero, where the overall termination does not change, there are more kinds of garbage that is not removed by total garbage collection. For example in `let` x = `...` `in` `seq` x x (using Haskell-notations) the subexpression `seq` x may be removed depending on the overall semantics of the whole program, but total garbage collection cannot achieve this using replacements by non-terminating constants.

A comparison between the space leaks using a practical garbage collection compared to total garbage collection shows, that for the three space-leaks above there is no difference. The impact of the non-optimality of total garbage collection is not strong: An analysis shows that also using an optimal garbage collector the copying of functions is a space leak and the idea can be transferred to the other space leaks above.

This chapter is based on (SSD19b).

Chapter 5

For CHF* we summarize time improvement results for sequential and parallel reduction strategies, where the definition of parallel evaluation is also given. This chapter is a summary of (SSSD18).

Chapter 6

For CHF* we first define the `size` of expressions. The measure *sps* is reused for the space measure *spmin* that is defined as the minimal *sps*-value of all successful evaluations.

Scheduling of parallel processes has a great impact on the overall space consumption *spmin*. Therefore we develop an algorithm that calculates the required space of multiple processes w.r.t. scheduling efficiently, if the processes are in sync at start and end points but apart from this run completely independent from each other – i.e. there are no further forms of synchronizations.

As abstract model for each process a list of integers represents the trace of space consumption of the process. We show that many of those integer values do not have an effect on the overall result and can be removed. Thus we give an algorithm for standardization that runs in linear time.

We then give an algorithm that takes the standard form and calculates the required space. Finally we define the algorithm SpOptN, that uses the algorithms above all together with minor additions and overall runtime $O((N+n)\log N)$ to calculate the required space of N processes, where n denotes the total input size.

Synchronizations are also analyzed using basic synchronization restrictions, that are Boolean conditions on simultaneous or relative time points of two processes. The complexity using such a set of Boolean constraints lead to an algorithm with complexity of $O(poly(n) \cdot n^{O(b \cdot N)})$, where $poly$ is a polynomial, b the number of Boolean conditions and n the total size of the input. However we see immediately that the runtime is polynomial if the number of Boolean conditions and the number of processors is fixed. The perfect partition problem is used to show that the problem is NP-complete if general synchronization restrictions are used.

We also show the relation to producer-consumer problems and to a variant of job shop scheduling.

The implementation of M1$_{sp}$ for LRP allowed complex and systematical analyses. Thus the abstract machines CIOM1$_{sp,int}$, CIOM1$_{sp,par}$ and CIOM1$_{t,par}$ are introduced for CHF*, that are based on the machine CIOM1.

The abstract machine CIOM1$_{sp,int}$ is the abstract machine CIOM1 extended by space measurement and garbage collection, where the space equivalence between CHF* and CIOM1$_{sp,int}$ is showed.

For the parallel case where synchronizations are not restricted, the calculation performed by an abstract machine to calculate the required space is complex. First of all the nondeterminism leads to different evaluations with different results and since the space optimum is the target, all combinations w.r.t. nondeterminism need to be considered. Moreover it may be a good idea to delay a thread as SpOptN above showed, therefore also in these cases many possible evaluations need to be calculated to find a space-optimal solution.

The delays of threads are made explicit using a so-called *delay-function* that decides, depending on the current situation (i.e. the next transition rule of the inspected thread), if it makes sense to delay a thread or not. We show that in many cases the delay of a thread does not have an effect on the overall result and therefore for a better runtime these delays should not be considered. Moreover threads waiting for resources are delayed per default, not increasing the overall solution space.

Intuitively we calculate a tree of machine states. It turns out that sometimes calculating the real tree can be helpful, since then parallelization can be used heavily. In the most cases we use an iterated depth first search where the stack contains the remaining nodes. Also the so far found leaf (if there already is one) with yet minimal space consumption requires bookkeeping, because this allows further optimizations: We show that in case a leaf is found, then all nodes on the stack can be removed that have a greater or equal so far calculated optimal space consumption. However we show that this has the effect, that a space-optimal leaf in the end is not guaranteed to be the runtime-optimal under the space-optimal leafs.

As further optimization checksums of machine states help to reduce the calculation time. Also α-equivalence testing can be used to improve the runtime in some cases. It turns out that all of these optimizations heavily depend on the analyzed program, sometimes the calculation of the real state tree is very fast due to the power of parallelization, however in the most cases the checksum-testing is the fastest method – the α-equivalence-testing itself has a high runtime and is only useful in specific cases where delays lead to many states that are identical but use different names.

The whole approach is implemented in the abstract machine $\text{CIOM1}_{sp,par}$. As already mentioned the space-optimal solution is not guaranteed to be the runtime-optimal under the space-optimal solutions, therefore using the same approach the abstract machine $\text{CIOM1}_{t,par}$ is implemented to find runtime-optimal solutions. The space equivalence between $\text{CIOM1}_{sp,par}$ and CHF^* using parallel evaluation and also the runtime equivalence between $\text{CIOM1}_{t,par}$ and CHF^* using parallel evaluation is showed.

Implementations of the abstract machines $\text{CIOM1}_{sp,par}$ and $\text{CIOM1}_{t,par}$ are then used for space and runtime analyses. Despite of the high runtime complexity of the abstract machines, the optimizations and configuration possibilities allow analyses of various scenarios.

We analyze the effect of parallelization of a calculation of a list that can be split up in parallel independent calculations, folding an addition over a tree comparing a pure and a parallelized implementation and an extension of common subexpression elimination to a thread-based variant. If the complexity of an analysis is too high, different configurations as stopping the calculation once a leaf is found or using no delays, can be used to calculate approximations that help to get oriented. If it is known that all threads run independently and the results of all are needed, then using no delays together with $\text{CIOM1}_{t,par}$ yields the optimal runtime.

The definition of space measurement for CHF^* and the results concerning SpOptN and synchronization restrictions above are based on (SSD19a). All results regarding the abstract machines $\text{CIOM1}_{sp,int}$, $\text{CIOM1}_{sp,par}$ and $\text{CIOM1}_{t,par}$ are not published earlier.

Zusammenfassung

In dieser Dissertation werden Speicherverbrauch und Laufzeit von lazy-auswertenden funktionalen Programmiersprachen untersucht. Dabei werden Analysen sowohl unter An- als auch Abwesenheit von Seiteneffekten in Form von Nebenläufigkeit durchgeführt.

Wir stellen die Inhalte der einzelnen Kapitel im Folgenden genauer vor.

Kapitel 2

In diesem Kapitel werden Grundlagen behandelt. Es werden die Kalküle LRP (SSS16b, SS15, SS14, SSS15b) und CHF* (SSSD18) definiert. LRP ist ein polymorph getypter lazy-auswertender Lambda-Kalkül, der um rekursive `let`-Ausdrücke, `seq`-Ausdrücke und `case`-Ausdrücke wie in Haskell erweitert wurde. CHF* verfügt über eine Ebene für monadische Funktionalitäten, wobei auf dieser Ebene Seiteneffekte möglich sind und eine funktionale Ebene, die mit LRP verglichen werden kann. CHF* ist eine leicht modifizierte Variante von CHF (SS11, SSS12), wir zeigen die semantische Äquivalenz der beiden Kalküle.

Die abstrakten Maschinen M1, IOM1 and CIOM1 werden definiert (siehe (Sab12)). Die abstrakte Maschine M1 ist kompatibel zu LRP und CIOM1 ist kompatibel zu CHF*.

Zudem zeigen wir eine häufig benutzte Beweistechnik mit Forking-Diagrammen und definieren oft genutzte Funktionen für LRP und CHF*.

Kapitel 3

Zunächst stellen wir den Kalkül LRPgc vor, welcher die sogenannte *Normalordnung* als operationale Semantik verwendet. LRPgc erweitert LRP um einen klassischen implementierbaren Garbage Collector, der nach jedem Schritt einer Normalordnungs-Reduktion ausgeführt wird. Dann wird die Größe `size` von Ausdrücken definiert und zudem das Platzmaß *spmax*, welches den Speicherverbrauch einer gesamten LRPgc-Programmausführung angibt, das heißt die größte `size` von Ausdrücken einer gesamten Normalordnungsreduktion, bei denen der Garbage Collector nichts entfernen kann.

Programm-Transformationen werden als Code-zu-Code-Transformationen aufgefasst, das heißt eine solche Transformation wird auf syntaktischer Ebene angewendet und

das Ergebnis der Transformation befindet sich ebenfalls weiterhin auf der syntaktischen Ebene.

Die Begriffe *Space Improvement* und *Space Equivalence* als Eigenschaften von Transformationen werden definiert. Die Intuition eines Space Improvements ist, dass die Transformation auf alle möglichen Unterausdrücke angewendet werden kann, bei denen das Pattern der Transformation übereinstimmt und in all diesen Fällen darf sich der Speicherverbrauch nicht erhöhen – für wenigstens einen Fall muss der Speicherverbrauch sinken und falls dies nicht der Fall ist, handelt es sich um eine Space Equivalence. Diese Definitionen benötigen Kontexte. Ein Kontext $\mathbb{C}$ ist ein Ausdruck mit einem einzigen Loch, in das Ausdrücke eingesetzt werden können, das heißt $\mathbb{C}[\cdot]$ mit $[\cdot]$ als Loch. Die starke Forderung, dass Space Improvements und Space Equivalences an jeder passenden Stelle angewendet werden können, wird mit Kontexten präzisiert:

Seien s und t LRPgc-Ausdrücke. s ist ein Space Improvement von t, falls die Ungleichung $\mathit{spmax}(\mathbb{C}[s]) \leq \mathit{spmax}(\mathbb{C}[t])$ für alle Kontexte $\mathbb{C}$ gilt. Die Definition für Space Equivalences ist analog, bloß dass die Bedingung $\mathit{spmax}(\mathbb{C}[s]) = \mathit{spmax}(\mathbb{C}[t])$ ist.

Die Anzahl der zu betrachtenden Kontexte für den Nachweis der Space-Improvement- oder Space-Equivalence-Eigenschaft einer bestimmten Transformation ist sehr hoch. Deshalb zeigen wir sogenannte Context Lemmas: Falls die Eigenschaft eines Space Improvements oder einer Space Equivalence für Reduktionskontexte (d.h. Kontexte welche die Call-by-Need-Strategie der Normalordnung modellieren) gilt, so gilt diese auch für alle Kontexte.

Für jede betrachtete Transformation T werden alle Überlappungen zwischen Normalordnung und T geprüft. Dazu werden Mengen von Forking-Diagrammen verwendet und dann Induktion angewendet um die entsprechende Eigenschaft nachzuweisen. Mithilfe der Context Lemmas kann die Anzahl an Forking-Diagrammen stark reduziert werden und somit wird der gesamte Beweis erleichtert. Auf diese Weise werden einige Space Improvements und Space Equivalences gezeigt.

Außerdem nennen wir eine Transformation *Space-Leak* wenn es mindestens einen Fall gibt, in dem der Speicherverbrauch unkontrollierbar ansteigt. Wir zeigen die Space-Leak-Eigenschaft der folgenden Transformationen: Das Kopieren von Funktionen, die Veränderung der Auswertungsreihenfolge unter Verwendung von Wissen über Striktheit und die Beseitigung von gemeinsamen Unterausdrücken.

Wir führen weitere Platz-Analysen durch, bei denen zusätzliches Wissen über den Speicherverbrauch vorhanden ist.

Außerdem führen wir die abstrakte Maschine $\mathrm{M1}_{sp}$ für Messungen des Speicherverbrauchs ein, welche eine Anpassung der abstrakten Maschine M1 ist. Wir benutzen eine Implementierung der abstrakten Maschine $\mathrm{M1}_{sp}$ für einige komplexere Platz- und Laufzeit-Analysen. Außerdem zeigen wir die Korrektheit der Platz-Messungen der M1. Eine solche Implementierung ermöglicht außerdem verschiedene Konfigurationen des Garbage Collectors und erlaubt somit Analysen des Einflusses von Garbage Collection.

Dieses Kapitel basiert auf (SSD18, SSD17).

Kapitel 4

Die oben erwähnten Analysen des Speicherverbrauchs in LRP sind abhängig vom verwendeten Garbage Collector. Wir führen die sogenannte *Total Garbage Collection* ein, um die Ergebnisse bezüglich Space Improvements und Space Equivalences unabhängiger vom verwendeten Garbage Collector zu halten.

Als Kalkül verwenden wir LR, das heißt LRP ohne Typen. Total Garbage Collection ersetzt einen Teilausdruck durch eine nicht-terminierende Konstante mit Größe Null, falls sich dadurch die gesamte Terminierungseigenschaft des Programms nicht verändert. Total Garbage Collection ist offensichtlich nicht implementierbar, da der Garbage Collector das Halteproblem eines Programms lösen muss, allerdings kann diese Art von Garbage Collection dennoch für Analysen des Speicherverbrauchs verwendet werden.

Das Maß `size` von LRP wird auf LR übertragen und außerdem definieren wir das Platzmaß *sps*, als die größte `size` von Ausdrücken, bei denen durch Total Garbage Collection nichts entfernt werden kann, die während der gesamten Normalordnungs-Reduktion vorkommen.

Die Begriffe *Total Space Improvement* und *Total Space Equivalence* als Eigenschaften von Transformationen werden definiert. Analog zu Space Improvements und Space Equivalences für LRP, kann ein Total Space Improvement auf alle möglichen Unterausdrücke angewendet werden, bei denen das Pattern der Transformation übereinstimmt und in all diesen Fällen darf sich der Speicherverbrauch nicht erhöhen.

Seien s und t LR-Ausdrücke. s ist ein Total Space Improvement von t, falls die Ungleichung $\mathit{sps}(\mathbb{C}[s]) \leq \mathit{sps}(\mathbb{C}[t])$ für alle Kontexte $\mathbb{C}$ gilt. Die Definition für Total Space Equivalences ist analog, mit der Bedingung $\mathit{sps}(\mathbb{C}[s]) = \mathit{sps}(\mathbb{C}[t])$.

Auch unter Verwendung von Total Garbage Collection ist die Anzahl an zu betrachtenden Kontexten für den Nachweis der Eigenschaften eines Total Space Improvements oder einer Total Space Equivalence zu hoch. Daher zeigen wir auch hier Context Lemmas: Falls die Eigenschaft eines Total Space Improvements oder einer Total Space Equivalence für Reduktionskontexte gilt, so gilt diese auch für alle Kontexte. Auf diese Weise werden einige Total Space Improvements und Total Space Equivalences gezeigt.

Der Begriff Space-Leak wird von LRP auf LR übertragen und genutzt, um die Space-Leak-Eigenschaft der folgenden Transformationen zu zeigen: Das Kopieren von Funktionen, die Veränderung der Auswertungsreihenfolge unter Verwendung von Wissen über Striktheit und die Beseitigung von gemeinsamen Unterausdrücken.

Wir ordnen Total Garbage Collection bezüglich Optimalität ein. Da Total Garbage Collection Unterausdrücke durch nicht-terminierende Konstanten mit einer `size` von Null ersetzt, wobei sich die gesamte Terminierungseigenschaft des Programms nicht ändern darf, kann mithilfe dieses Ansatzes nicht jede Form von Garbage gelöscht werden. Zum Beispiel kann im Ausdruck `let` $x = \ldots$ `in seq` x x (unter Verwendung von Haskell-Schreibweisen) der Teilausdruck `seq` x gelöscht werden, wenn die Semantik des gesamten Programms dies zulässt, allerdings kann Total Garbage Collection das mit einer Ersetzung durch eine nicht-terminierende Konstante nicht erreichen, da diese in dem Fall ein terminierendes Programm zu einem nicht-terminierenden umwandeln würde.

Ein Vergleich zwischen Space-Leaks unter Verwendung eines implementierbaren Garbage Collectors mit Total Garbage Collection zeigt, dass dies keinen Einfluss auf die Space-Leak-Eigenschaften der drei oben gezeigten Space-Leak-Transformationen hat. Eine Analyse zeigt außerdem, dass selbst ein optimaler Garbage Collector im Falle des Kopierens von Funktionen an der Space-Leak Eigenschaft dieser Transformation nichts ändern kann. Dieser Ansatz kann auf die anderen oben genannten Space-Leaks übertragen werden.

Diese Kapitel basiert auf (SSD19b).

Kapitel 5

Für CHF* fassen wir Ergebnisse von Laufzeitoptimierungen sowohl unter Verwendung von sequentieller als auch paralleler Auswertung zusammen, wobei auch die parallele Auswertungsstrategie für CHF* definiert wird.

Dieses Kapitel ist eine Zusammenfassung von (SSSD18).

Kapitel 6

Für CHF* definieren wir die Größe `size` von Ausdrücken. Das Maß *sps* wird wiederverwendet für das Platzmaß *spmin*, das als minimaler *sps*-Wert aller erfolgreichen Auswertungen definiert ist.

Das Scheduling von parallelen Prozessen hat einen starken Einfluss auf den gesamten Speicherverbrauch *spmin*. Daher entwickeln wir einen Algorithmus, der den benötigten Speicherplatz mehrerer Prozesse bezüglich Scheduling effizient berechnet, sofern diese am Anfang und Ende synchron und ansonsten unabhängig voneinander sind.

Als abstraktes Modell benutzen wir für jeden Prozess eine Liste von Zahlen, welche zeitlich geordnet den Speicherverbrauch des Prozesses angibt. Wir zeigen, dass einige dieser Zahlen keinen Einfluss auf das Gesamtergebnis haben und entfernt werden können. Folglich entwickeln wir einen Algorithmus, welcher in linearer Laufzeit diese Form von Standardisierung durchführt.

Dann entwerfen wir einen Algorithmus, der eine solche Standardform erwartet und den benötigten Speicherplatz berechnet. Schließlich definieren wir den Algorithmus SpOptN, welcher die obigen Algorithmen mit kleinen Ergänzungen anwendet und mit einer Gesamtlaufzeit in $O((N + n) \log N)$ den benötigten Speicherplatz von N Prozessen berechnet, wobei n die gesamte Eingabegröße bezeichnet.

Auch Synchronisierungen zwischen Prozessen werden analysiert. Dazu werden grundlegende Beschränkungen in Form von einfachen booleschen Formeln verwendet, die absolute oder relative Aussagen bezüglich Synchronität zweier Zeitpunkte erlauben.

Die Verwendung solcher booleschen Formeln führt zu einem Algorithmus mit Laufzeit $O(poly(n) \cdot n^{O(b \cdot N)})$, wobei $poly$ ein Polynom, b die Anzahl an booleschen Bedingungen and n die gesamte Eingabegröße ist. Jedoch sehen wir sofort, dass die Laufzeit polynomiell ist, falls die Anzahl der booleschen Beschränkungen und die Anzahl der Prozessoren konstant ist. Das Perfect-Partition-Problem kann verwendet werden um die NP-Vollständigkeit für den Fall beliebiger Synchronisierungs-Beschränkungen zu zeigen.

Zudem zeigen wir den Bezug zu Producer-Consumer-Problemen und einer Variante von Job-Shop-Scheduling.

Die Implementierung der abstrakten Maschine $\mathrm{M1}_{sp}$ für LRP ermöglicht komplexe und systematische Analysen. Daher werden die abstrakten Maschinen $\mathrm{CIOM1}_{sp,int}$, $\mathrm{CIOM1}_{sp,par}$ und $\mathrm{CIOM1}_{t,par}$ für CHF* eingeführt, welche auf der CIOM1 basieren.

Die abstrakte Maschine $\mathrm{CIOM1}_{sp,int}$ ist die abstrakte Maschine CIOM1 erweitert um Platzmessung und Garbage Collection. Wir zeigen die Äquivalenz bezüglich der Platzmessungen zwischen CHF* und $\mathrm{CIOM1}_{sp,int}$.

Eine abstrakte Maschine ist komplex, falls im parallelen Fall jede Art von Synchronisierung möglich ist. Zunächst führt der Nicht-Determinismus zu unterschiedlichen Ausführungsreihenfolgen mit unterschiedlichen Ergebnissen und da der optimale Speicherverbrauch das Ziel ist, müssen alle durch Nicht-Determinismus verursachte Kombinationen betrachtet werden. Außerdem zeigt SpOptN, dass eine Verzögerung eines Threads den Speicherverbrauch verringern kann, daher müssen auch für diesen Fall viele mögliche Auswertungen betrachtet werden um eine platzoptimale Lösung zu finden.

Das Verzögern von Threads wird durch eine sogenannte *Delay-Funktion* umgesetzt, die entsprechend zur aktuellen Situation (d.h. der nächste Schritt des betrachteten Threads) entscheidet, ob es sinnvoll ist, einen Thread zu verzögern oder nicht. Für viele Fälle können wir zeigen, dass eine Verzögerung eines Threads keinen Einfluss auf das Gesamtergebnis hat und deshalb sollten solche Verzögerungen aufgrund der besseren Laufzeit außer Acht gelassen werden. Außerdem werden auf Ressourcen wartende Threads automatisch verzögert, somit wird der Lösungsraum nicht vergrößert.

Intuitiv betrachtet wird ein Baum von Maschinen-Zuständen berechnet. Es hat sich herausgestellt, dass in einigen Fällen die Berechnung des wirklichen Baums hilfreich ist, da dann viel parallelisiert werden kann. In den meisten Fällen wird eine iterative Tiefensuche verwendet, wobei der Stack die verbleibenden Knoten enthält. Außerdem wird – falls vorhanden – das soweit gefundene Blatt mit bisher minimalem Speicherverbrauch gespeichert, da dadurch weitere Optimierungen ermöglicht werden: Wir zeigen, dass im Falle eines gefundenen Blattes, alle Knoten des Stacks mit größerem oder gleichem soweit berechneten optimalen Platzverbrauch gelöscht werden können. Wir zeigen jedoch, dass deshalb ein berechnetes platzoptimales Blatt nicht zwingend eine optimale Laufzeit unter den platzoptimalen Blättern hat.

Checksummen von Maschinen-Zuständen werden als weitere Optimierung genutzt um die Berechnungszeit zu verbessern. Auch α-Äquivalenz kann ausgenutzt werden

um die Laufzeit in manchen Fällen zu verbessern. Es stellt sich heraus, dass all diese Optimierungen sehr stark vom zu analysierenden Programm abhängen, manchmal ist die Berechnung des richtigen Baums aufgrund der Parallelisierung sinnvoll, jedoch ist in den meisten Fällen die iterative Tiefensuche mit Checksummen-Optimierung die schnellste Variante – das Testen der α-Äquivalenz hat selbst eine hohe Laufzeit und ist nur sinnvoll in Fällen, in denen Verzögerungen zu vielen semantisch identischen Zuständen führen, die sich nur durch die Benennung von Variablen unterscheiden.

Der gesamte Ansatz ist in der abstrakten Maschine $\mathrm{CIOM1}_{sp,par}$ umgesetzt. Wie schon erwähnt hat die platzoptimale Lösung nicht zwingend auch die beste Laufzeit unter den platzoptimalen Lösungen, daher wird unter Verwendung des gleichen Ansatzes die abstrakte Maschine $\mathrm{CIOM1}_{t,par}$ eingeführt, um die optimale Laufzeit zu berechnen. Die Kompatibilität der Platzmessungen von $\mathrm{CIOM1}_{sp,par}$ und CHF* unter Verwendung paralleler Auswertung und ebenso die Kompatibilität der Laufzeitmessungen zwischen $\mathrm{CIOM1}_{t,par}$ and CHF* unter Verwendung paralleler Auswertung werden gezeigt.

Implementierungen der abstrakten Maschinen $\mathrm{CIOM1}_{sp,par}$ und $\mathrm{CIOM1}_{t,par}$ werden verwendet um Speicher- und Laufzeit-Analysen durchzuführen. Trotz der hohen Laufzeit-Komplexität der abstrakten Maschinen, erlauben die Optimierungen und Konfigurationsmöglichkeiten verschiedene Analysen.

Wir analysieren den Einfluss von Parallelisierung auf die Berechnung einer Liste, die in mehrere parallele und unabhängige Berechnungen aufgespalten werden kann. Ebenso wird die Anwendung einer fold-Funktion unter Verwendung eines Additionsoperators auf einen Baum analysiert, wobei eine klassische und eine parallelisierte Variante verglichen werden. Außerdem wird eine Erweiterung der Eliminierung von gemeinsamen Unterausdrücken auf Thread-Ebene betrachtet.

Falls die Komplexität einer Analyse zu hoch ist, können verschiedene Konfigurationen verwenden werden, wie z.B. Rückgabe des zuerst gefundenen Blattes oder das Abschalten jeglicher künstlicher Verzögerungen durch die Delay-Funktion. Auf diese Weise werden Approximationen berechnet die zur Orientierung benutzt werden können. Falls bekannt ist, dass alle Thread unabhängig sind und die Ergebnisse aller Threads benötigt werden, dann benötigt man keine durch Delay-Funktion ausgelösten Verzögerungen und die abstrakte Maschine $\mathrm{CIOM1}_{t,par}$ liefert die optimale Laufzeit.

Die obige Definition der Speicherplatzmessung für CHF* und die Ergebnisse bezüglich SpOptN und Synchronisierungsbeschränkungen basieren auf (SSD19a). Alle genannten Ergebnisse zu den abstrakten Maschinen $\mathrm{CIOM1}_{sp,int}$, $\mathrm{CIOM1}_{sp,par}$ und $\mathrm{CIOM1}_{t,par}$ wurden nicht zuvor veröffentlicht.

Acknowledgments

I would like to thank everyone who supported me during the writing of this thesis.

I thank Prof. Dr. Manfred Schmidt-Schauß for the excellent collaboration over the last years. Without the several constructive and fruitful discussions and advices this work would not be possible.

I thank Prof. Dr. Nils Bertschinger for his interest in my thesis and being an examiner.

I thank Prof. Dr. David Sabel. His constructive advices lead to new ideas and helped to improve this thesis a lot. I am also very thankful for his support in my first months as doctorand.

I thank Prof. Dr. Lars Hedrich for his support in many different areas over the last years.

I thank my doctoral colleague Yunus Kutz. We had many interesting discussions and we managed a lot of work for the professorship.

I am very grateful to all colleagues during the last years for the great atmosphere.

I thank my parents Heike Dallmeyer and Uwe Dallmeyer for their support all the time, allowing me to study and to write this thesis.

Finally I would like to thank my love Somayeh Hassanzadeh. Her lovely support was very important to write this thesis.

Contents

1

Introduction

In this chapter we first motivate and introduce the topic of research, give a relation to other work, provide an overview of the results of this work and outline the thesis.

1.1 Motivation

In modern days the importance of communication and availability of information has a great impact on the daily life. This leads to a lot of challenges w.r.t. scalability and concurrency. In the last decade most of the classical relational databases were replaced by different variants of non-relational databases (e.g. see (HJ11, DCL18)).

On the other hand the usage of smartphones increased massively (RMZ12) leading to new challenges, since the memory of smartphones is often restricted and also a high demand on processors drains the battery. Hence for smartphones it is a benefit if an application has a good runtime while not using too much memory to achieve such a runtime.

Moreover the runtime strongly increases if algorithms make heavy use of random accesses on memory and the memory is not sufficient. In such cases the algorithms need to be adapted, so that they run efficiently using external memory such as hard disks or SSDs (for example see (San98, BMV13, HMP$^+$18)). The more internal memory is available, the more can be kept in internal memory – this motivates to keep the internal memory consumption low, so that the mentioned approaches using external memory are not required or only for larger amounts of data.

Compilers usually make heavy use of runtime-optimizations, since this often allows to write code that is easy to read and simple to maintain. However runtime-optimizations may lead to a high increase of space consumption. In such cases further analyses are needed, to find out whether the increase of space consumption is acceptable in relation to the improvement of runtime.

This motivates to improve and analyze the space consumption of programs, where space consumption refers to internal memory. In this theses we focus on the area of lazy evaluating functional languages as Haskell. Especially for calculations free of side effects, parallelization often can be implemented easily (e.g. see (HMPJ05, CLPJ$^+$07, LRS$^+$03, MNJ11, PS09, PGF96, MBCP14)), hence Haskell-programs have the potential to be scaled using a network of processors. In the following section we give a brief introduction into this area and show relations to existing work.

1.2 Survey

Lazy evaluation allows a declarative way of programming, since the dependencies between different data is important but the exact evaluation sequence does not need to be specified by the programmer (see (PGF96, Hug89, LOMPnM05)).

As base for analyses of correctness, runtime and space consumption, it helps to have a calculus that can be seen as a core language of the target programming language. The practical target language is Haskell, therefore calculi based on lazy variants of the lambda calculus are useful.

Syntactically a lambda calculus consists of variables, anonymous functions (often called abstractions) and applications (AFM+95, MSC99), e.g. the identity function can be represented by the anonymous function $\lambda x.x$, where x is a variable. A lazy lambda calculus implements a call-by-name (Abr90) reduction strategy. However call-by-name may duplicate work and therefore languages as Haskell use call-by-need (AFM+95) using memoization to implement sharing and avoid duplication of work. All calculi used in this thesis are based on a lazy lambda calculus using a call-by-need reduction strategy. For concurrency extra additions are needed.

The lazy lambda calculus as described above does not have any kind of datatypes. Hence for practical analyses and a higher expressiveness it is important, to extend the lazy lambda calculus by datatypes as in Haskell. Also explicit recursion is easier than using fixed point combinators (see (PJ87)) as in a classic lambda calculus. Furthermore monomorphic or polymorphic typing rules are often an useful extension, since changes to a program that do not preserve typing are usually not interesting in practice.

The used calculi in this thesis are variants of LRP (SSS16b, SS15, SS14, SSS15b) and CHF (SS11, SSS12), where LRP can be used to analyze pure Haskell-like programs and CHF for programs that are comparable to Concurrent Haskell (PGF96):

LRP is a lazy lambda calculus extended by recursive `let`-expressions, `seq`-expressions (to force strictness at some positions) and `case`-expressions as in Haskell. Moreover the calculus provides polymorphic typing.

CHF (SS11, SSS12) follows a two-layered approach as Concurrent Haskell (PGF96): An action layer to implement monadic features that might introduce side-effects and a functional layer that is comparable to LRP. As termination-property may- and should-convergence is used (SS11, SSS10), where may-convergence states that a process reduces to a successful process and should-convergence is the property that a process remains may-convergent after reductions.

Moreover it is often useful to define abstract machines for a practical execution of programs, where Sestoft's abstract machine (Ses97) is a solid base for different calculi. For LRP and CHF the abstract machines from (Sab12), that are extended versions of (Ses97), can be used with adaptions for space analyses.

To analyze the correctness, runtime and space behavior of a program, we use source-to-source comparisons. This means that a given source of a program s is modified on some position by a transformation T resulting in s' and then we can check if s and s'

are semantically equivalent or if s' has a lower or equal runtime than s and similar for space consumption.

In the area of correctness there are many results, for example in (San95b, SSS08b, SS07, SSS08a, SSSH09, SSS15a, SSS17, SSSD18). Also in the area of runtime improvements a lot of results exists as in (San91, San95a, San97, MS99, SSS15a, SSS16a). For space consumption the results are rare, in (GS99, GS01, Gus01) the approach of (San95a, MS99) for runtime is adapted to space using an abstract machine semantics based on the abstract machine of (Ses97). They use an untyped language and the space measurement differs and is rather complex compared to our space measurement. (HH19) introduces an approach that is suitable for less complex languages, however adding constructs like recursive `let`-expressions and `seq`-expressions to the language, the complexity seems to increase rapidly.

An important property of a compiler optimization is its correctness. Hence we require that a runtime or space improving transformation does not change the semantics of a program. In most of the cases we can import the correctness results before we start analyzing the space consumption of programs. In this thesis the notion of correctness of (SSS08b) is used, since it can be used directly for our calculi in this thesis:

A context is a program of the corresponding programming language with a single hole at some position. We now put the compared programs s and s' into every context and if for all contexts the termination property is equivalent (for CHF both may- and should-convergence are considered), then s and s' are called contextual equivalent. For runtime analyses in (SSS15a, SSS17) a similar approach is used where for all contexts the runtime is not allowed to increase.

Intuitively considering contexts is the same as to plug s and s' in arbitrary greater programs and the correctness and improvement property w.r.t. runtime or space consumption holds. In most cases a context lemma is proved, that allows to conclude that, if for a subset of all contexts the specified property (e.g. correctness, runtime-improvement or space-improvement) holds, the property holds for all contexts.

1.3 Results

For space measurement the LRP-variant LRPgc is defined in Definition 3.2, where an implementable garbage collection after each reduction step is applied. Proposition 3.1 shows the semantic equivalence of LRP and LRPgc.

Matching the intuition of transformations that improve or maintain the overall space consumption, the notions *space improvement* and *space equivalence* are defined (see Definition 3.7).

For LRPgc we show two context lemmas, that reduce the amount of cases that need to be considered during the proofs of space improvements and equivalences: Lemma 3.6 for space improvements and Lemma 3.7 for space equivalences.

Theorem 3.14 shows, that several transformations are space improvements or space equivalences and for a few transformations the increase of space consumption and the circumstances are analyzed. Also the space behavior for special cases are analyzed as in Proposition 3.11.

The abstract machine $\mathrm{M1}_{sp}$ is defined in Definition 3.19. This machine is a variant of Sestoft's machine (Ses97) extended by space measurement capabilities and Theorem 3.15 shows that this abstract machine performs a correct space measurement. An implementation of $\mathrm{M1}_{sp}$ is used for several analyses of more complex programs.

Using LR as calculus, that is LRP without types, a total garbage collector is defined in Definition 4.2, that replaces all unneeded subexpressions of an expression by a non-terminating constant with size 0. This garbage collector is not correct and also not implementable, since termination needs to be decided to find all garbage collectable positions, but it is suitable to perform space analyses. Definition 4.5 defines the space measurement and requires that total garbage collection is applied whenever possible.

The notions of *total space improvement* and *total space equivalence* are defined (see Definition 4.6), with the intuition that a transformation improves or maintains the space consumption under total garbage collection. Two context lemmas are showed: Lemma 4.5 for total space improvements and Lemma 4.6 for total space equivalences. Theorem 4.14 shows, that several transformations are total space improvements or total space equivalences and also for a few transformations that they might increase the space consumption.

Since the total garbage collector replaces subexpressions by non-terminating expressions, even this non-implementable garbage collector is not able to catch every kind of garbage. Proposition 4.7 shows that a certain transformation is even not a space improvement if a theoretical optimal garbage collector is used.

For CHF*, a convergence-equivalent variant of CHF, the notions of sequential and parallel time-improvements (see Definition 5.1 and Definition 5.3) are defined and several time improvements are shown in Section 5.2.

The space measurement for CHF* is defined in Definition 6.3. For analyses of space-optimal schedules, we introduce an abstract model only consisting of lists of numbers in Definition 6.4, where each number corresponds to the current space consumption of the corresponding process. Also we give an algorithm that calculates the required space w.r.t. scheduling, if the processes are independent with the only requirement that they are in sync at the start and end points (Algorithm 6.3). The correctness of this algorithm and complexity $O((N+n)\log N)$, where n is the total size of the input and N the number of processes, is proved in Theorem 6.3.

Theorem 6.4 shows, that the use of combinations of Boolean conditions on simultaneous or relative time points of two processes used for synchronizations lead to an algorithm with complexity $O(poly(n) \cdot n^{O(b \cdot N)})$, where $poly$ is a polynomial, b the number of the Boolean conditions and n the total size of the input. For a fixed number of processes and Boolean conditions the same algorithm runs in polynomial time (see Corollary 6.1). For general synchronization restrictions the problem is NP-complete as shown in Theorem 6.5.

The abstract machine CIOM1$_{sp,int}$ is defined in Definition 6.18 and can be used for space analyses in CHF* if a sequential reduction order is used (i.e. a scheduler chooses a single thread to proceed, but no threads proceed in parallel). Theorem 6.6 shows that the space measurement of CIOM1$_{sp,int}$ coincides with the space measurement of CHF* using a sequential reduction order.

Moreover Definition 6.29 yields the abstract machine CIOM1$_{sp,par}$, where threads can proceed in parallel and therefore all interleavings are considered for the calculation of the required space. This yields a tree representing all interleavings, where often subtrees can be cut off as shown in Lemma 6.7 and also duplicated calculation-states can be removed using checksum and α-equivalence testing as shown in Lemma 6.8. The space equivalence between CIOM1$_{sp,par}$ and CHF* using parallel evaluation (as defined in Definition 5.2) is shown in Theorem 6.7.

Since Proposition 6.7 shows that the result calculated by CIOM1$_{sp,par}$ is space-optimal, but not guaranteed to be the runtime-optimal under the space-optimal, the abstract machine CIOM1$_{t,par}$ is defined, to calculate the minimal runtime if parallel evaluation is used. The compatibility of runtime-measurement between CIOM1$_{t,par}$ and CHF* using parallel evaluation is shown in Theorem 6.8.

Implementations of the mentioned abstract machines and algorithm SpOptN are used for several analyses w.r.t. space consumption and runtime.

Our space measurement using an eager implementable garbage collector and also using the total garbage collector, turns out to be effective and allows easier analyses than (GS99, GS01, Gus01). Moreover our results apply to a more powerful calculus than used in (HH19). In the area of scheduling the runtime is usually optimized and therefore SpOptN is a first step into the area of space-optimal schedules. The abstract machines CIOM1$_{sp,par}$ and CIOM1$_{t,par}$ provide a tool for space and runtime analyses in the concurrent scenario. The enhancement of space analyses to a concurrent scenario extend the work of Gustavsson and Sands (GS99, GS01, Gus01) and the results of Section 5.2 can be seen as extensions of (LM99) in a wider sense.

1.4 Outline

In the following we give an outline of this thesis.

Chapter 2 recalls the calculi LRP (as defined in (SSS16b, SS15, SS14, SSS15b)) and CHF* (as defined in (SSSD18)) and corresponding abstract machines.

LRP is defined in Section 2.1 and CHF* is defined in Section 2.2. In these sections for each calculus syntax, typing rules and operational semantics are given as well as the notions of contextual equivalence, that allows reasoning about the correctness of program transformations. The equivalence of CHF and CHF* is also shown.

Section 2.3 gives a brief overview of the translation of simple Haskell-subprograms to LRP and CHF* and also defines datatype declarations comparable to Haskell. In Section 2.4 we recall three abstract machines (as defined in (SS15, Sab12)) that can be used for LRP and CHF* and are the base of further abstract machines.

Section 2.5 introduces a proof technique that is often used in this thesis and Section 2.6 defines several needed LRP-functions or functions useable on the functional layer of CHF*.

In **Chapter 3** (based on (SSD18, SSD17)) space analyses for LRP are performed.

Section 3.1 defines the LRP-variant LRPgc that is compatible to LRP and applies a practical implementable garbage collection whenever possible.

In Section 3.2 the notions *space improvement*, *space equivalence* and *space leak* are defined. In Section 3.3 context lemmas for space improvement and space equivalence are proved for LRPgc. Section 3.4 contains analyses of several transformations w.r.t. space consumption using the context lemmas. Hence for many transformations it is shown in this section, that they are space improvements, space equivalences or space leaks.

Section 3.5 contains analyses of examples where the space usage is controlled, i.e. specific knowledge can be used.

In Section 3.6 the abstract machine $M1_{sp}$ is introduced and the compatibility to LRPgc is shown. Moreover this section contains analyses of examples using an implementation of $M1_{sp}$.

In **Chapter 4** (based on (SSD19b)) space analyses for LR (i.e. LRP without types) are performed, where a non-implementable garbage collector is used. Section 4.1 defines total garbage collection, a space measurement where total garbage collection is applied before measuring and also the notions of *total space improvements* and *total space equivalences.*

In Section 4.2 context lemmas for total space improvement and total space equivalence are proved for LR. Section 4.3 contains analyses of several transformations w.r.t. space consumption using the context lemmas. Hence for many transformations it is shown in this section, that they are total space improvements, total space equivalences or space leaks.

Section 4.4 compares total garbage collection with optimal garbage collection. Also for a certain transformation we show that it is a space leak even if a theoretical optimal garbage collector is used.

Chapter 5 is a summary of results and needed definitions for runtime analyses of (SSSD18).

Section 5.1 defines parallel evaluation for CHF*. Also the notions of *sequential time improvements* and *parallel time improvements* and similar for equivalences are defined.

Section 5.2 presents the results of several proven time improvements for CHF*, both for sequential and parallel evaluation.

In **Chapter 6** space analyses for CHF* are performed. Section 6.1 (based on (SSD19a)) defines the space measurement for CHF*.

In Section 6.2 (based on (SSD19a)) an abstract model for parallel processes is introduced. The algorithm SpOptN is then developed to calculate the required space w.r.t. scheduling for independent processes, where only start- and end-points are required to be in sync. Also the addition of Boolean conditions on simultaneous or relative time points of two processes and impact on the runtime complexity is analyzed.

In Section 6.3 we introduce the abstract machines CIOM1$_{sp,int}$ for space analyses using sequential evaluation, CIOM1$_{sp,par}$ for space analyses using parallel evaluation and CIOM1$_{t,par}$ for runtime analyses using parallel evaluation. The correctness of measurement is shown for all of these abstract machines and then implementations of the machines are used for analyses of examples.

We conclude and discuss potential future work in **Chapter 7**.

2

Lazy Evaluating Calculi

The two calculi *Polymorphically Typed Lazy Lambda Calculus (LRP)* and *Concurrent Haskell with Futures (CHF)* are used for the later work in the field of improvements. LRP is used for deterministic analyses while CHF is used in the (nondeterministic) concurrent scenario. Also abstract machines for both calculi are defined.

2.1 Polymorphically Typed Lazy Lambda Calculus

The calculus Polymorphically Typed Lazy Lambda Calculus (LRP) (SSS16b, SS15, SS14, SSS15b) is a lazy lambda calculus (for example see (AFM⁺95),(MSC99)) extended by $\mathtt{case}_K$-expressions for every constructor-type K, `seq`-expressions $(\mathtt{seq}\ s\ t)$, recursive `letrec`-expressions, polymorphic abstractions $\Lambda a.s$ for polymorphic functions and type applications $(s\ \tau)$ for type-instantiations. The syntax of types and expressions is given in the following definition:

Definition 2.1 (LRP Syntax of Types and Expressions)
Let type variables $a, a_i \in \mathit{TVar}$ and term variables $x, x_i \in \mathit{Var}$. Every type constructor K has an arity $ar(K) \geq 0$ and a finite set D_K of data constructors $c_{K,i} \in D_K$ with an arity $ar(c_{K,i}) \geq 0$.
Types *Typ* and term variables *PTyp* are defined as follows:

$$\begin{array}{lcl} \tau \in \mathit{Typ} & ::= & a \mid (\tau_1 \to \tau_2) \mid (K\ \tau_1\ \dots\ \tau_{ar(K)}) \\ \rho \in \mathit{PTyp} & ::= & \tau \mid \forall a.\rho \end{array}$$

Expressions *Expr* are generated by this grammar with $n \geq 1$ and $k \geq 0$:

$$\begin{array}{lcl} s, t, s_i \in \mathit{Expr} & ::= & u \mid x :: \rho \mid (s\ \tau) \mid (s\ t) \mid (\mathtt{seq}\ s\ t) \mid (c_{K,i} :: (\tau)\ s_1\ \dots\ s_{ar(c_{K,i})}) \\ & & \mid (\mathtt{case}_K\ s\ \mathtt{of}\ \{(\mathit{Pat}_{K,1} \to t_1)\ \dots\ (\mathit{Pat}_{K,|D_K|} \to t_{|D_K|})\}) \\ & & \mid (\mathtt{letrec}\ x_1 :: \rho_1 = s_1, \dots, x_n :: \rho_n = s_n\ \mathtt{in}\ t) \\ \mathit{Pat}_{K,i} & ::= & (c_{K,i} :: (\tau)\ (x_1 :: \tau_1)\ \dots\ (x_{ar(c_{K,i})} :: \tau_{ar(c_{K,i})})) \\ u \in \mathit{PExpr} & ::= & (\Lambda a_1.\Lambda a_2.\dots\Lambda a_k.\lambda x :: \tau.s) \end{array}$$

Note that `seq` is used to force the evaluation of the first argument before returning the second argument. Especially in combination with a `letrec`-environment this can be used to implement strict evaluation for certain subexpressions.

The `case`-alternatives must have exactly one alternative $((c_{K,i}\ x_1\ \dots\ x_{ar(c_{K,i})}) \to s_i)$ for every constructor $c_{K,i}$ of type K, where the variables $x_1, \dots, x_{ar(c_{K,i})}$ occurring in the pattern $(c_{K,i}\ x_1\ \ldots\ x_{ar(c_{K,i})})$ are pairwise distinct and become bound with scope s_i. Often $\{x_{g(i)} = s_{f(i)}\}_{i=j}^{m}$ is used as abbreviation for $x_{g(j)} = s_{f(j)}, \ldots, x_{g(m)} = s_{f(m)}$. Also E is used as abbreviation for `letrec`-environments, that are a multiset of (recursive) bindings of the form $x = s$. $alts$ is an abbreviation for `case`-alternatives. $FV(s)$ and $BV(s)$ is used to denote free and bound variables of an expression s, $LV(E)$ to denote the binding variables of a `letrec`-environment and $(c_{K,i}\ s_1\ \ldots\ s_{ar(c_{K,i})})$ is often abbreviated with $c\ \overrightarrow{s}$ and $\lambda x_1.\ldots\lambda x_n.s$ with $\lambda x_1,\ldots,x_n.s$. Moreover we often write c instead of $c_{K,i}$.

LRP provides polymorphic typing of `letrec`-binding-variables. Hence this is not the full polymorphic typing as used in Haskell and we also do not have type classes, but the typing of LRP is strong enough for polymorphic lists and functions working on such data structures.

Typing is important, since programs as `map id xs` written in Haskell-notion, that maps the identity functions to the list `xs` is only equivalent to `xs` under typing, but in an untyped scenario `xs` is not required to be a list and in this case both expressions are not contextual equivalent. Thus we look at well-typed programs and this eases the proofs and theorems a lot. An LRP-program is called *well-typed* if it can be typed using the rules given in the following definition:

Definition 2.2 (LRP Typing Rules)

$$\frac{s :: \rho}{\Lambda a.s :: \forall a.\rho} \qquad \frac{s :: \tau_1 \quad \forall i : Pat_i :: \tau_1 \quad \forall i : t_i :: \tau_2}{(\texttt{case}_K\ s\ \texttt{of}\ \{\ (Pat_1 \to t_1)\ \ldots\ (Pat_{|D_K|} \to t_{|D_K|})\ \}) :: \tau_2}$$

$$\frac{s :: \tau_2}{(\lambda x :: \tau_1.s) :: \tau_1 \to \tau_2} \qquad \frac{s :: \forall a.\rho}{(s\ \tau) :: \rho[\tau/a]} \qquad \frac{s :: \tau_1 \to \tau_2 \quad t :: \tau_1}{(s\ t) :: \tau_2} \qquad \frac{s :: \tau \quad t :: \tau'}{(\texttt{seq}\ s\ t) :: \tau'}$$

$$\frac{\begin{array}{c} s_1 :: \tau_1,\ \ldots,\ s_{ar(c)} :: \tau_{ar(c)} \quad \tau = \tau_1 \to \cdots \to \tau_{ar(c)} \to \tau_{ar(c)+1} \\ type(c) = \forall a_1,\ldots,a_m.\tau'' \quad \exists \tau'_1,\ldots,\tau'_m : \tau''[\tau'_1/a_1,\ldots,\tau'_m/a_m] = \tau \end{array}}{(c :: \tau\ s_1\ \ldots\ s_{ar(c)}) :: \tau_{ar(c)+1}}$$

$$\frac{s_1 :: \rho_1\ \ldots\ s_n :: \rho_n \quad t :: \rho}{(\texttt{letrec}\ x_1 :: \rho_1 = s_1,\ldots,x_n :: \rho_n = s_n\ \texttt{in}\ t) :: \rho}$$

Most of the typing rules are quite straightforward, but the rule for constructor applications is a little bit more complicated, since this rule checks, whether the whole constructor application is an instance of the annotated (polymorphic) type at the constructor name.

Note that these rules require that all names of Λ-bound type variables are disjoint (distinct variable convention for type variables). Otherwise the typing rules would not recognize that y has two different type variables assigned to it as in this example:

$$(\Lambda a.\lambda z :: a.\texttt{letrec}\ y :: a = y :: a\ \texttt{in}\ \Lambda a.\lambda x :: a.y :: a)$$

However, for better readability we often often omit the type labels.

As an operational semantics the so-called *normal order reduction* is used. This is a small-step reduction relation that implements a call-by-need-strategy using a rewriting approach. First we give a definition of contexts:

Definition 2.3 (Context and Multicontext)

1. A context $\mathbb{C} \in$ *CCtxt* is an expression with exactly one hole $[\cdot]$ at expression position. $\mathbb{C}[e]$ denotes the result of replacing the hole in $\mathbb{C}$ by expression e.
2. A multicontext M is an expression with zero or more (different) holes at expression positions.

Now we give an algorithm that calculates the reduction position of an LRP-expression.

Algorithm 2.1 (LRP Labeling Algorithm)
Let s and t be LRP-expressions. The input expression s is labeled with top, written as s^{top}. Now apply the rules from below as long as possible. If a fail occurs then no reduction position exists, otherwise return the found position. The label $a \vee b$ means a or b. ${}^{\mathrm{top}}$ means reduction of the top expression, ${}^{\mathrm{sub}}$ of a subexpression, ${}^{\mathrm{vis}}$ marks already visited subexpressions and ${}^{\mathrm{nontarg}}$ is used for visited variables which are not target of (cp)-reductions (see Definition 2.5).

$$\begin{array}{lcl}
(s\ t)^{\mathrm{sub}\vee\mathrm{top}} & \to & (s^{\mathrm{sub}}\ t)^{\mathrm{vis}} \\
(s\ \tau)^{\mathrm{sub}\vee\mathrm{top}} & \to & (s^{\mathrm{sub}}\ \tau)^{\mathrm{vis}} \\
(\mathtt{letrec}\ E\ \mathtt{in}\ s)^{\mathrm{top}} & \to & (\mathtt{letrec}\ E\ \mathtt{in}\ s^{\mathrm{sub}})^{\mathrm{vis}} \\
(\mathtt{letrec}\ x = s, E\ \mathtt{in}\ C[x^{\mathrm{sub}}]) & \to & (\mathtt{letrec}\ x = s^{\mathrm{sub}}, E\ \mathtt{in}\ C[x^{\mathrm{vis}}]) \\
(\mathtt{letrec}\ x = s, y = C[x^{\mathrm{sub}}], E\ \mathtt{in}\ t) & \to & (\mathtt{letrec}\ x = s^{\mathrm{sub}}, y = C[x^{\mathrm{vis}}], E\ \mathtt{in}\ t) \\
 & & \text{if } C \neq [\cdot] \\
(\mathtt{letrec}\ x = s, y = x^{\mathrm{sub}}, E\ \mathtt{in}\ t) & \to & (\mathtt{letrec}\ x = s^{\mathrm{sub}}, y = x^{\mathrm{nontarg}}, E\ \mathtt{in}\ t) \\
(\mathtt{seq}\ s\ t)^{\mathrm{sub}\vee\mathrm{top}} & \to & (\mathtt{seq}\ s^{\mathrm{sub}}\ t)^{\mathrm{vis}} \\
(\mathtt{case}_K\ s\ \mathtt{of}\ alts)^{\mathrm{sub}\vee\mathrm{top}} & \to & (\mathtt{case}_K\ s^{\mathrm{sub}}\ \mathtt{of}\ alts)^{\mathrm{vis}} \\
(\mathtt{letrec}\ x = s^{\mathrm{vis}\vee\mathrm{nontarg}}, y = C[x^{\mathrm{sub}}], & \to & \text{Fail} \\
\quad E\ \mathtt{in}\ t) & & \\
(\mathtt{letrec}\ x = C[x^{\mathrm{sub}}], E\ \mathtt{in}\ t) & \to & \text{Fail}
\end{array}$$

The labeling algorithm follows a top-down-approach, where all demanded subexpressions are marked and in the end the reduction position is calculated. The label ${}^{\mathrm{top}}$ in rule 3 ensures that the algorithm does not look further than a single step into `letrec`-expressions. The algorithm has linear runtime in the syntactical size of the given LRP-expression. Note that practical implementations use abstract machines, that perform the search for a reduction position without extra runtime.

For different kinds of context classes we use the following notions:

Definition 2.4 (Context Classes *RCtxt*, *SCtxt*, *TCtxt*)

1. *Reduction context* $\mathbb{R} \in$ *RCtxt*: A context such that its hole is labeled with ${}^{\mathrm{top}}$ or ${}^{\mathrm{sub}}$ after applying Algorithm 2.1.
2. *Surface context* $\mathbb{S} \in$ *SCtxt*: A context where the hole is not in an abstraction.
3. *Top context* $\mathbb{T} \in$ *TCtxt*: A surface context where the hole is not in a `case`-alternative.

(SSS08b) also gives a definition of reduction contexts by context free grammars.

Reduction contexts are for example $[\cdot]$, $([\cdot]\ e)$, $(\mathtt{case}\ [\cdot] \ldots)$ and $\mathtt{letrec}\ x = [\cdot], y = x, \ldots\ \mathtt{in}\ (\mathtt{seq}\ x\ \mathtt{True})$. Reduction contexts are surface- as well as top-contexts. A value is an abstraction $\lambda x.s$, a polymorphic abstraction u or a constructor application $c\ \overrightarrow{s}$.

Now we can define basic reduction rules of LRP which need the labels calculated with Algorithm 2.1.

Definition 2.5 (Basic LRP Reduction Rules)

(lbeta) $((\lambda x.s)^{\text{sub}}\ r) \to (\texttt{letrec}\ x = r\ \texttt{in}\ s)$

(Tbeta) $((\Lambda a.u)^{\text{sub}}\ \tau) \to u[\tau/a]$

(cp-in) $(\texttt{letrec}\ x_1 = v^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}\ C[x_m^{\text{vis}}])$
$\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}\ C[v])$
where v is a polymorphic abstraction

(cp-e) $(\texttt{letrec}\ x_1 = v^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E, y = C[x_m^{\text{vis}}]\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x_1 = v, \{x_i = x_{i-1}\}_{i=2}^{m}, E, y = C[v]\ \texttt{in}\ r)$
where v is a polymorphic abstraction

(llet-in) $(\texttt{letrec}\ E_1\ \texttt{in}\ (\texttt{letrec}\ E_2\ \texttt{in}\ r)^{\text{sub}}) \to (\texttt{letrec}\ E_1, E_2\ \texttt{in}\ r)$

(llet-e) $(\texttt{letrec}\ E_1, x = (\texttt{letrec}\ E_2\ \texttt{in}\ t)^{\text{sub}}\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ E_1, E_2, x = t\ \texttt{in}\ r)$

(lapp) $((\texttt{letrec}\ E\ \texttt{in}\ t)^{\text{sub}}\ s) \to (\texttt{letrec}\ E\ \texttt{in}\ (t\ s))$

(lcase) $(\texttt{case}_K\ (\texttt{letrec}\ E\ \texttt{in}\ t)^{\text{sub}}\ \texttt{of}\ alts)$
$\to (\texttt{letrec}\ E\ \texttt{in}\ (\texttt{case}_K\ t\ \texttt{of}\ alts))$

(lseq) $(\texttt{seq}\ (\texttt{letrec}\ E\ \texttt{in}\ s)^{\text{sub}}\ t) \to (\texttt{letrec}\ E\ \texttt{in}\ (\texttt{seq}\ s\ t))$

(seq-c) $(\texttt{seq}\ v^{\text{sub}}\ t) \to t$ if v is a value

(seq-in) $(\texttt{letrec}\ x_1 = (c\ \vec{s})^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}\ C[(\texttt{seq}\ x_m^{\text{vis}}\ t)])$
$\to (\texttt{letrec}\ x_1 = (c\ \vec{s}), \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}\ C[t])$

(seq-e) $(\texttt{letrec}\ x_1 = (c\ \vec{s})^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E, y = C[(\texttt{seq}\ x_m^{\text{vis}}\ t)]\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x_1 = (c\ \vec{s}), \{x_i = x_{i-1}\}_{i=2}^{m}, E, y = C[t]\ \texttt{in}\ r)$

(case-c) $(\texttt{case}_K\ c^{\text{sub}}\ \texttt{of}\ \{\dots(c \to t)\dots\}) \to t$ if $ar(c) = 0$, otherwise:
$(\texttt{case}_K\ (c\ \vec{s})^{\text{sub}}\ \texttt{of}\ \{\dots((c\ \vec{x}) \to t)\dots\})$
$\to (\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{ar(c)}\ \texttt{in}\ t)$

(case-in) Let y_i be fresh variables. If $ar(c) = 0$:
$(\texttt{letrec}\ x_1 = c^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}$
$C[(\texttt{case}_K\ x_m^{\text{vis}}\ \texttt{of}\ \{(c \to r)\dots\})])$
$\to (\texttt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}\ C[r])$
otherwise:
$(\texttt{letrec}\ x_1 = (c\ \vec{t})^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}$
$C[(\texttt{case}_K\ x_m^{\text{vis}}\ \texttt{of}\ \{((c\ \vec{z}) \to r)\dots\})])$
$\to (\texttt{letrec}\ x_1 = (c\ \vec{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^{m}, E\ \texttt{in}$
$C[\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \texttt{in}\ r])$

(case-e) Let y_i be fresh variables. If $ar(c) = 0$:
$(\texttt{letrec}\ x_1 = c^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[(\texttt{case}_K\ x_m^{\text{vis}}\ \texttt{of}\ \{(c \to r_1)\dots\})],$
$E\ \texttt{in}\ r_2)$
$\to (\texttt{letrec}\ x_1 = c, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[r_1], E\ \texttt{in}\ r_2)$
otherwise:
$(\texttt{letrec}\ x_1 = (c\ \vec{t})^{\text{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m},$
$u = C[(\texttt{case}_K\ x_m^{\text{vis}}\ \texttt{of}\ \{((c\ \vec{z}) \to r)\dots\})], E\ \texttt{in}\ s)$
$\to (\texttt{letrec}\ x_1 = (c\ \vec{y}), \{y_i = t_i\}_{i=1}^{ar(c)}, \{x_i = x_{i-1}\}_{i=2}^{m},$
$u = C[\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{ar(c)}\ \texttt{in}\ r], E\ \texttt{in}\ s)$

We go through the basic reduction rules of Definition 2.5:

In contrast to the classical β-reduction, that is a substitution

$$((\lambda x.s)\ r) \to s[r/x]$$

(lbeta) shares the variable x by the introduction of a `letrec`-expression and therefore avoids duplication of work.

(Tbeta) is needed for the type-instantiation of polymorphic types. The rules (cp-in) and (cp-e) copy abstractions which are needed when the reduction rules have to reduce an application $(f\ a)$ where f is an abstraction defined in a `letrec`-environment. (llet-in) and (llet-e) merge nested `letrec`-expressions, while (lapp), (lcase) and (lseq) move a `letrec`-expression out of an application, `case`-expression and `seq`-expression.

(seq-in) and (seq-e) evaluate `seq`-expressions, where the first argument points to a `letrec`-binding that contains a value. (case-c), (case-in) and (case-e) evaluate `case`-expressions, where the instantiation of pattern-variables for the appropriate `case`-alternative is implemented using new `letrec`-expressions.

Many rules are related to each other since they apply the same idea on different places. This leads to the following definition, that defines families of closely related transformations:

Definition 2.6 (Basic LRP-Reduction Rules Families)

1. (cp) is the union of (cp-in) and (cp-e).
2. (llet) is the union of (llet-in) and (llet-e).
3. (lll) is the union of (lapp), (lcase), (lseq) and (llet).
4. (case) is the union of (case-c), (case-in) and (case-e).
5. (seq) is the union of (seq-c), (seq-in) and (seq-e).

Normal order reduction steps are defined as follows:

Definition 2.7 (Reduction Steps)

1. Reduction step $s \xrightarrow{LRP} t$: If Algorithm 2.1 terminates successfully on s then one applicable rule of Definition 2.5 yields t.
2. $s \xrightarrow{LRP,*} t$: s reduces to t with an arbitrary nonnegative number of reduction steps.
3. $s \xrightarrow{LRP,+} t$: s reduces to t with at least one reduction step.
4. $s \xrightarrow{LRP,k} t$: s reduces to t with exactly k reduction steps where $k \geq 0$.

Now the termination of an LRP-expression can be defined:

Definition 2.8 (Weak Head Normal Form, Convergence and Divergence)

1. A weak head normal form (WHNF) is a value, or an expression `letrec` E `in` v, where v is a value, or an expression `letrec` $x_1 = c\ \overrightarrow{t}, \{x_i = x_{i-1}\}_{i=2}^{m}, E$ `in` x_m.
2. An expression s converges to an expression t ($s{\downarrow}t$ or $s{\downarrow}$ if we do not need t) if $s \xrightarrow{LRP,*} t$ where t is a WHNF.
3. An expression s diverges ($s{\uparrow}$) if there is no $s \xrightarrow{LRP,*} t$ where t is a WHNF.
4. The constant-symbol $\perp$ represents a closed diverging expression.
   E.g. `letrec x = x in x`.

Now we give a formal definition for program transformations:

Definition 2.9 (Program Transformation)
A program transformation P is binary relation on LRP-expressions. We write $s \xrightarrow{P} t$, if $(s,t) \in P$. For a set of contexts X and a transformation P, the transformation (X,P) is the closure of P w.r.t. the contexts in X, i.e. $\mathbb{C}[s] \xrightarrow{X,P} \mathbb{C}[t]$ iff $\mathbb{C} \in X$ and $s \xrightarrow{P} t$. Note that we write $s \xrightarrow{i,P} t$ if P is not applied in normal order.

We also give a definition of α-equivalence of LRP-expressions and LRP-types, with the intuition, that a program constructed by renaming bound variables without interfering with free variables remains semantic equivalent.

Definition 2.10 (α-Equivalence)
Let s and s' be LRP-expressions and $t, t' \in$ *PTyp*.

1. An α-renaming step $s \rightarrow_{\alpha,e} s'$ for expressions replaces a bound variable of s, i.e. $x \in BV(s)$, by a variable $y \notin BV(s) \cup FV(s)$.
2. An α-renaming step $t \rightarrow_{\alpha,t} t'$ for types is defined similar to $\rightarrow_{\alpha,e}$, but bound variables are type variables that are bound by a $\boldsymbol{\lambda}$.
3. $=_{\alpha,e}$ is the reflexive-transitive-closure of $\rightarrow_{\alpha,e}$ and $=_{\alpha,t}$ is the reflexive-transitive-closure of $\rightarrow_{\alpha,t}$. For simplicity we only want to write $=_\alpha$ for both in the future. Which one is meant is clear from the context.

Note that the α-equivalence is needed for the assumption, that the distinct variable convention is fulfilled at any time during reduction sequences. For instance (lbeta) needs to introduce new variable names to avoid name conflicts.

As notion of correctness we use contextual equivalence:

Definition 2.11 (Contextual Preorder and Equivalence)
Let s and t be LRP-expressions.

1. Contextual preorder $\leq_c$: $\quad s \leq_c t \iff \forall C[\cdot] : C[s]\downarrow \Rightarrow C[t]\downarrow$.
2. Contextual equivalence $\sim_c$: $\quad s \sim_c t \iff s \leq_c t \wedge t \leq_c s$.

A program transformation P is *correct* iff $P \subseteq \sim_c$.

Since the termination-property needs to be shown for each context, for programs with different semantics a context can be found, where the termination differs.

Example 2.1 (Contextual Equivalence)
Let s = `True` and t = `False`.
For $C = [\cdot]$ we have $C[s]\downarrow$ and $C[t]\downarrow$.
But for C = `case` $[\cdot]$ `of` $\{$(`True` → `True`) (`False` → `letrec` $x = x$ `in` x)$\}$, we have $C[s]\downarrow$ and $C[t]\uparrow$, hence s and t are not contextual equivalent.

The normal order reduction is correct:

Proposition 2.1 (Correctness of Normal-Order-Reduction)
As shown in (SSS08b) the transformations (lbeta), (cp), (lll), (seq), (case) are correct.

Note that the normal-order-reduction is type-safe, i.e. each expression that is a result of a normal-order-reduction-step can be typed using the rules in Definition 2.2.

The following example shows the normal-order-reduction of a little program, where the sharing implemented by (lbeta), copying of abstractions by (cp) and the use of seq-expressions is demonstrated.

Example 2.2 (Reduction Sequence of LRP-program)

```
letrec neg = (λb.case b of {(True → False) (False → True)}), x = (neg True)
    in seq x (neg x)
```

$\xrightarrow{\text{(cp-e)}}$

```
letrec neg = (λb.case b of {(True → False) (False → True)}),
         x   = ((λb.case b of {(True → False) (False → True)}) True),
    in seq x (neg x)
```

$\xrightarrow{\text{(lbeta)}}$

```
letrec neg = (λb.case b of {(True → False) (False → True)}),
         x   = letrec b_1 = True in (case b_1 of {(True → False) (False → True)}),
    in seq x (neg x)
```

$\xrightarrow{\text{(llet-e)}}$

```
letrec neg = (λb.case b of {(True → False) (False → True)}),
         x   = case b_1 of {(True → False) (False → True)},
         b_1 = True
    in seq x (neg x)
```

$\xrightarrow{\text{(case-e)}}$

```
letrec neg = (λb.case b of {(True → False) (False → True)}),
         x   = False, b_1 = True
    in seq x (neg x)
```

$\xrightarrow{\text{(seq-in)}}$

```
letrec neg = (λb.case b of {(True → False) (False → True)}),
         x   = False, b_1 = True
    in (neg x)
```

$\xrightarrow{\text{(cp-in)}}$

```
letrec neg = (λb.case b of {(True → False) (False → True)}),
         x   = False, b_1 = True
    in ((λb.case b of {(True → False) (False → True)}) x)
```

$\xrightarrow{\text{(lbeta)}}$

```
letrec neg = (λb.case b of {(True → False) (False → True)}),
         x   = False, b_1 = True
    in (letrec b_2 = x in case b_2 of {(True → False) (False → True)})
```

$\xrightarrow{\text{(llet-in)}}$

```
letrec neg = (λb.case b of {(True → False) (False → True)}),
         x   = False, b_1 = True, b_2 = x
    in case b_2 of {(True → False) (False → True)}
```

$\xrightarrow{\text{(case-in)}}$

```
letrec neg = (λb.case b of {(True → False) (False → True)}),
         x   = False, b_1 = True, b_2 = x
    in True
```

The following measure is used for runtime:

Definition 2.12 (Reduction Length Measures rln, rlnall **and** rln_{LCSC}**)**
Let s be a closed LRP-expression with $s{\downarrow}s_0$ and $LCSC := \{\text{(lbeta)}, \text{(cp)}, \text{(case)}, \text{(seq)}\}$.

1. $\text{rln}(s)$ is the sum of all (lbeta)-, (case)- and (seq)-reduction steps in $s{\downarrow}s_0$.
2. $\text{rlnall}(s)$ is the number of all reduction steps in $s{\downarrow}s_0$.
3. $\text{rln}_{LCSC}(s)$ is the number of all a-reduction steps with $a \in LCSC$ in $s{\downarrow}s_0$.

The measure is based on the number of specific normal-order-reduction-steps and focuses on certain transformations, that have a great impact on the (practical) runtime.

In summary LRP can be seen as a core language of Haskell with polymorphic types in `letrec`-bindings. The use of typing is motivated, since we consider correct program transformations in later chapters and the omission of untyped programs often reduces the amount of cases to consider.

LR is LRP without types (SSS16b): Let ε : LRP → LR be defined as $\varepsilon((s\ \tau)) := \varepsilon(s)$, $\varepsilon(\Lambda a.s) := \varepsilon(s), \varepsilon(x :: \rho) := x$ and $\varepsilon(c :: \rho) := c$. LR also omits (TBeta). Then $\varepsilon(s) \sim_c \varepsilon(t)$ $\Rightarrow s \sim_c t$ but caused by the typed `case`, $s \sim_c t$ does not imply $\varepsilon(s) \sim_c \varepsilon(t)$.

2.2 Concurrent Haskell with Futures

Concurrent Haskell was proposed in (PGF96) and implemented with slightly a few changes in the Glasgow Haskell Compiler (Pey01, Wad95, PS09). The basic idea uses two layers: A pure functional layer, where only deterministic calculations are performed and an action layer, that implements monadic programming features (see (PW93, Wad95, Pey01)) and allows actions that might introduce side-effects, e.g. the creation of threads and a common storage for multiple threads.

The calculus Concurrent Haskell with Futures (CHF*) (SSSD18) (semantic equivalent to CHF defined in (SS11, SSS12)) follows the same two-layered approach, where the pure functional layer is comparable to LRP (see Section 2.1).

Definition 2.13 (CHF* Syntax of Types, Processes and Expressions)
Let term variables $x,x_i \in Var$. Every type constructor K has an arity $ar(K) \geq 0$ and a finite set D_K of data constructors $c_{K,i} \in D_K$ with an arity $ar(c_{K,i}) \geq 0$. Let $n \geq 1$. Types *Typ* are defined as follows:

$$\tau \in \mathit{Typ} \quad ::= \quad \texttt{IO}\ \tau \mid (K\ \tau_1 \ldots \tau_{ar(K)}) \mid \texttt{MVar}\ \tau \mid \tau_1 \to \tau_2$$

Expressions *Expr* and monadic expressions *MExpr* are generated by this grammar:

$$\begin{array}{lcl} s,t,s_i \in \mathit{Expr} & ::= & m \mid x \mid (\lambda x.s) \mid (s\ t) \mid (\texttt{seq}\ s\ t) \mid (c_{K,i}\ s_1 \ldots s_{\mathrm{ar}(c_{K,i})}) \\ & & \mid (\texttt{case}_K\ s\ \texttt{of}\ \{(\mathit{Pat}_{K,1} \to t_1)\ \ldots\ (\mathit{Pat}_{K,|D_K|} \to t_{|D_K|})\}) \\ & & \mid (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) \\ \mathit{Pat}_{K,i} & ::= & (c_{K,i}\ x_1\ \ldots\ x_{ar(c_{K,i})}) \\ m \in \mathit{MExpr} & ::= & \texttt{return}\ s \mid s \gg\!= t \mid \texttt{future}\ s \mid \texttt{takeMVar}\ s \mid \texttt{newMVar}\ s \\ & & \mid \texttt{putMVar}\ s\ t \end{array}$$

Processes *Proc* are defined as follows:

$$P \in \mathit{Proc} \quad ::= \quad (P_1 \mid P_2) \mid x \Leftarrow s \mid \nu x.P \mid x\,\mathbf{m}\,s \mid x\,\mathbf{m}\,- \mid x = s$$

$(\texttt{IO}\ \tau)$ stands for a monadic action with return-type τ and $(\texttt{MVar}\ \tau)$ stands for an `MVar`-reference with content type τ.

Parallel processes are implemented by parallel composition $\mid$. The scope of variables can be restricted using the ν-binders. Moreover we write $x \Leftarrow s$ for a concurrent thread that binds the result of the evaluation of s to the variable x, where x is also called the *future* x (FF99, NSSSS07, NSS06). Each process has a main-thread, i.e. $x \overset{\text{main}}{\Longleftarrow} s$. The common storage of threads is implemented using mutable variables named *MVars*. An *MVar* x can be filled with an expression s, denoted as $x\,\mathbf{m}\,s$, or is empty, i.e. $x\,\mathbf{m}\,-$. Moreover a global heap contains shared expressions, i.e. consisting of bindings $x = s$ from binding variables to expressions. An introduced variable x of a Process P is a future, a name of an MVar or a binding variable. Introduced variables are visible to the whole process, except their scope is restricted by a ν-binder. For instance in $\nu x.P$ the variable x is only visible for P. We call a process *well-formed*, if all introduced variables are pairwise distinct and there exists at most one main-thread.

Functional expressions can be variables, abstractions $(\lambda x.s)$, applications $(s\ t)$, `seq`-expressions $(\texttt{seq}\ s\ t)$, constructor applications $(c\ s_1 \dots s_{\text{ar}(c)})$, $\texttt{case}_K$-expressions for every constructor-type K and recursive `letrec`-expressions.

The `case`-alternatives must have exactly one alternative $((c_{K,i}\ x_1\ \dots\ x_{ar(c_{K,i})}) \to s_i)$ for every constructor $c_{K,i}$ of type K, where the variables $x_1, \dots, x_{ar(c_{K,i})}$ occurring in the pattern $(c_{K,i}\ x_1\ \dots\ x_{ar(c_{K,i})})$ are pairwise distinct and become bound with scope s_i. Often $\{x_{g(i)} = s_{f(i)}\}_{i=j}^{m}$ is used as abbreviation for $x_{g(j)} = s_{f(j)}, \dots, x_{g(m)} = s_{f(m)}$. Also E is used as abbreviation for `letrec`-environments, that are a multiset of (recursive) bindings of the form $x = s$. $alts$ is an abbreviation for `case`-alternatives. *FV*(s) and *BV*(s) is used to denote free and bound variables of an expression s (*FV*(P) and *BV*(P) for a process P resp.), *LV*(E) to denote the binding variables of a `letrec`-environment and $(c_{K,i}\ s_1\ \dots\ s_{ar(c_{K,i})})$ is often abbreviated with $c\ \vec{s}$ and $\lambda x_1. \dots \lambda x_n.s$ with $\lambda x_1, \dots, x_n.s$. Moreover we often write c instead of $c_{K,i}$.

Monadic expressions are operators that create MVars using `newMVar`, access MVars implemented by `takeMVar` and `putMVar`, perform a sequential composition of IO-operations using ≫= (the bind-operator), create new threads using `future` where the idea is to bind the result of the thread to the specified variable, or lift an expression to a monadic expression implemented by `return`.

A *functional value* is either an abstraction or constructor application. A *monadic value* is a monadic expression of one of the following forms: $(\texttt{return}\ s)$,$(s \gg= t)$,$(\texttt{future}\ s)$, $(\texttt{newMVar}\ s)$, $(\texttt{takeMVar}\ s)$ or $(\texttt{putMVar}\ s\ t)$. A *value* is either a functional or monadic value.

We also give a definition of α-equivalence of CHF*-expressions, with the intuition, that a program constructed by renaming bound variables without interfering with free variables remains semantically equivalent. Note that we do need a definition of α-equivalence for types, since α-equivalence of CHF*-types is the same as syntactical equality caused by the absence of type-variables in the monomorphic type-setting.

Definition 2.14 (α-Equivalence)

1. Let s and s' be CHF*-expressions. An α-renaming step $s \to_\alpha s'$ for expressions replaces a bound variable of s, i.e. $x \in BV(s)$, by a variable $y \notin BV(s) \cup FV(s)$.
2. $=_\alpha$ is the reflexive-transitive-closure of $\to_\alpha$.

Structural congruence is used to equate obviously equal processes and is defined as the least congruence satisfying the equations:

Definition 2.15 (Structural Congruence ≡)

1. $P_1 \mid P_2 \equiv P_2 \mid P_1$
2. $(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$
3. $(\nu x.P_1) \mid P_2 \equiv \nu x.(P_1 \mid P_2)$ if $x \notin FV(P_2)$
4. $\nu x_1.\nu x_2.P \equiv \nu x_2.\nu x_1.P$
5. $P_1 \equiv P_2$ if $P_1 =_\alpha P_2$

A CHF*-program is called *well-typed* if it can be typed using the typing rules given in the following definition:

Definition 2.16 (CHF* Typing Rules)

$$\frac{\Gamma \vdash s :: \texttt{IO}\ \tau}{\Gamma \vdash x \Leftarrow s :: \texttt{wt}} \qquad \frac{\Gamma \vdash s :: \tau}{\Gamma \vdash x = s :: \texttt{wt}} \qquad \frac{\Gamma \vdash P_1 :: \texttt{wt} \quad \Gamma \vdash P_2 :: \texttt{wt}}{\Gamma \vdash P_1 \mid P_2 :: \texttt{wt}}$$

$$\frac{\Gamma(x) = \texttt{MVar}\ \tau \quad \Gamma \vdash s :: \tau}{\Gamma \vdash x\, \mathbf{m}\, s :: \texttt{wt}} \qquad \frac{\Gamma(x) = \texttt{MVar}\ \tau}{\Gamma \vdash x\, \mathbf{m} - :: \texttt{wt}} \qquad \frac{\Gamma \vdash s :: \tau}{\Gamma \vdash \texttt{newMVar}\ s :: \texttt{IO}\ (\texttt{MVar}\ \tau)}$$

$$\frac{\Gamma \vdash s :: \tau}{\Gamma \vdash \texttt{return}\ s :: \texttt{IO}\ \tau} \qquad \frac{\Gamma \vdash s :: \texttt{IO}\ \tau_1 \quad \Gamma \vdash t :: \tau_1 \to \texttt{IO}\ \tau_2}{\Gamma \vdash s \gg\!= t :: \texttt{IO}\ \tau_2} \qquad \frac{\Gamma \vdash s :: \texttt{IO}\ \tau}{\Gamma \vdash \texttt{forkIO}\ s :: \texttt{IO}\ \tau}$$

$$\frac{\Gamma \vdash s :: \texttt{MVar}\ \tau}{\Gamma \vdash \texttt{takeMVar}\ s :: \texttt{IO}\ \tau} \qquad \frac{\Gamma \vdash s :: \texttt{MVar}\ \tau \quad \Gamma \vdash t :: \tau}{\Gamma \vdash \texttt{putMVar}\ s\ t :: \texttt{IO}\ ()} \qquad \frac{\Gamma \vdash P :: \texttt{wt}}{\Gamma \vdash \nu x.P :: \texttt{wt}}$$

$$\frac{\forall i : \Gamma \vdash s_i :: \tau_i \quad \tau_1 \to \ldots \to \tau_n \to \tau_{n+1} \in types(c)}{\Gamma \vdash (c\ s_1\ \ldots\ s_{\mathrm{ar}(c)}) :: \tau_{n+1}} \qquad \frac{\Gamma \vdash s :: \tau_1 \to \tau_2 \quad \Gamma \vdash t :: \tau_1}{\Gamma \vdash (s\ t) :: \tau_2}$$

$$\frac{\Gamma(x) = \tau_1 \quad \Gamma \vdash s :: \tau_2}{\Gamma \vdash (\lambda x.s) :: \tau_1 \to \tau_2} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau} \qquad \frac{\begin{array}{c}\Gamma \vdash s :: \tau_1 \quad \Gamma \vdash t :: \tau_2 \\ \tau_1 = \tau_3 \to \tau_4 \text{ or } \tau_1 = (K \ldots)\end{array}}{\Gamma \vdash (\texttt{seq}\ s\ t) :: \tau_2}$$

$$\frac{\Gamma \vdash s :: \tau_1 \text{ and } \tau_1 = (K\ \ldots) \quad \forall i : \Gamma \vdash (c_i\ x_{1,i}\ \ldots\ x_{n_i,i}) :: \tau_1 \quad \forall i : \Gamma \vdash s_i :: \tau_2}{\Gamma \vdash (\texttt{case}_K\ s\ \texttt{of}\ \{(c_1\ x_{1,1}\ \ldots\ x_{n_1,1} \to t_1)\ \ldots\ (c_m\ x_{1,m}\ \ldots\ x_{n_m,m} \to t_m)\}) :: \tau_2}$$

$$\frac{\forall i : \Gamma(x_i) = \tau_i \quad \forall i : \Gamma \vdash s_i :: \tau_i \quad \Gamma \vdash t :: \tau}{\Gamma \vdash (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) :: \tau}$$

Let Γ be the global typing function. For instance $\Gamma(x)$ yields the type for x where we assume that every variable has a built-in type. To express that a type τ can be derived for an expression s using Γ, we write $\Gamma \vdash e :: \tau$. On the process-level we write $\Gamma \vdash P :: \texttt{wt}$ if Γ is able to derive a type for process P, hence P is called well-typed.

The type system is monomorphic for simplicity, but we assume that data constructors of every type have a polymorphic type as usual, whereas the constructors in the language are monomorphic. We write $types(c)$ for the set of monomorphic types of constructor c and let $()$ be the unit type. Also note that we require the distinct variable convention, hence each bound variable is required to have disjoint names – otherwise bound and independent variables with the same name might overlap in Γ.

As an operational semantics the so-called *standard reduction* is used. This is a small-step reduction relation that implements a call-by-need-strategy using a rewriting approach. At first we define the notion of successful processes:

Definition 2.17 (Successful Process)
A well-formed process P is *successful*, if P has a main-thread of the form $x \overset{\text{main}}{\Longleftarrow} \texttt{return}\ s$, i.e. $P \equiv \nu x_1 \dots \nu x_n.(x \overset{\text{main}}{\Longleftarrow} \texttt{return}\ s \mid P')$

We only define the standard reduction for well-formed processes that are not successful. The reason is that parsers can capture non well-formed processes and in Haskell all threads terminate, if the main-thread terminates. Contexts are defined as follows:

Definition 2.18 (Context and Multicontext)
1. A context $\mathbb{C} \in$ *CCtxt* is a process or an expression with exactly one hole $[\cdot]$.
   We assume that the hole $[\cdot]$ is typed and carries a type label, which we sometimes write as $[\cdot^\tau]$.
   For a context $\mathbb{C}[\cdot^\tau]$ and an expression $s :: \tau$ we write $\mathbb{C}[s]$ for the result of replacing the hole in $\mathbb{C}$ with s.
2. A multicontext M is a process or an expression with zero or more (different) holes.

For the different syntactical categories different contexts are needed:

Definition 2.19 (Contexts Classes *ECtxt*, *MCtxt*, *FCtxt*, *PCtxt*, *LCtxt* and $\widehat{LCtxt}$)

$$\begin{array}{lcl}
\mathbb{E} \in \mathit{ECtxt} & ::= & [\cdot] \mid (\mathbb{E}\ e) \mid \texttt{case}\ \mathbb{E}\ \texttt{of}\ alts \mid \texttt{seq}\ \mathbb{E}\ e \\
\mathbb{M} \in \mathit{MCtxt} & ::= & [\cdot] \mid \mathbb{M} \gg\!= e \\
\mathbb{F} \in \mathit{FCtxt} & ::= & \mathbb{E} \mid \texttt{takeMVar}\ \mathbb{E} \mid \texttt{putMVar}\ \mathbb{E}\ e \\
\mathbb{D} \in \mathit{PCtxt} & ::= & [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D} \\
\mathbb{L} \in \mathit{LCtxt} & ::= & x \Leftarrow \mathbb{M}[\mathbb{F}] \mid (x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \dots \mid x_2 = \mathbb{E}_2[y] \mid y = \mathbb{E}_1) \\
 & & \quad \text{s.t. } \mathbb{E}_i \neq [\cdot] \text{ for } 2 \leq i \leq n \\
\widehat{\mathbb{L}} \in \widehat{\mathit{LCtxt}} & ::= & x \Leftarrow \mathbb{M}[\mathbb{F}] \mid (x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \dots \mid x_2 = \mathbb{E}_2[y] \mid y = \mathbb{E}_1) \\
 & & \quad \text{s.t. } \mathbb{E}_i \neq [\cdot] \text{ for } 1 \leq i \leq n
\end{array}$$

For expressions usual call-by-name *expression evaluation contexts ECtxt* are used. The *monadic contexts MCtxt* yield the next monadic action in a sequence of monadic actions. Often `takeMVar` and `putMVar` require that the first argument is evaluated, this is modeled by the *forcing contexts FCtxt*. The contexts *LCtxt* are used to find the redex of an already selected thread, where the next reduction may be a monadic or functional evaluation that might require to go through a chain of bindings. For copying, we require that for variable-to-variable bindings variables are copied (not only the end of a variable chain as in LRP), implemented by $\widehat{LCtxt}$, since for `takeMVar`- and `putMVar`-operations the real variable name is needed for the access of the MVar.

The context classes in Definition 2.19 are important for the definition of the evaluation of CHF*-programs, but especially for the work in the area of improvements we also use the following two context classes:

Definition 2.20 (Context Classes *SCtxt* and *TCtxt*)
1. *Surface context* $\mathbb{S} \in$ *SCtxt*: A context where the hole is not in an abstraction.
2. *Top context* $\mathbb{T} \in$ *TCtxt*: A surface context where the hole is not in a `case`-alternative.

The basic reduction rules of CHF* are given in the following definition, where the outer process context is omitted.

Definition 2.21 (Basic CHF Reduction Rules)

(lunit)	$y \Leftarrow \mathbb{M}[\texttt{return}\ s \gg= t] \rightarrow y \Leftarrow \mathbb{M}[t\ s]$
(nmvar)	$y \Leftarrow \mathbb{M}[\texttt{newMVar}\ s] \rightarrow \nu x.(y \Leftarrow \mathbb{M}[\texttt{return}\ x] \mid x\,\mathbf{m}\,s)$ where x is fresh
(tmvar)	$y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\,\mathbf{m}\,s \rightarrow y \Leftarrow \mathbb{M}[\texttt{return}\ s] \mid x\,\mathbf{m}\,-$
(pmvar)	$y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ s] \mid x\,\mathbf{m}\,- \rightarrow y \Leftarrow \mathbb{M}[\texttt{return}\ ()] \mid x\,\mathbf{m}\,s$
(fork)	$y \Leftarrow \mathbb{M}[\texttt{future}\ s] \rightarrow \nu z.(y \Leftarrow \mathbb{M}[\texttt{return}\ z] \mid z \Leftarrow s)$ where z is fresh
(unIO)	$y \Leftarrow \texttt{return}\ s \rightarrow y = s$ if the thread is not the main-thread
(lbeta)	$\mathbb{L}[((\lambda x.s)\ t)] \rightarrow \nu x.(\mathbb{L}[s] \mid x = t)$
(cp)	$\widehat{\mathbb{L}}[x] \mid x = v \rightarrow \widehat{\mathbb{L}}[v] \mid x = v$ if v is an abstraction or a variable
(cpcxa)	$\widehat{\mathbb{L}}[x] \mid x = (c\ s_1 \ldots\ s_n)$
	$\rightarrow \nu y_1, \ldots y_n.(\widehat{\mathbb{L}}[x] \mid x = (c\ y_1\ \ldots\ y_n) \mid y_1 = s_1 \mid\ \ldots\ \mid y_n = s_n)$
	if c is a constructor, `return`, $\gg=$, `takeMVar`, `putMVar`, `newMVar`,
	or `future` and in addition some s_i is not a variable.
	Only the non-variables s_j are abstracted.
(cpcxb)	$\widehat{\mathbb{L}}[x] \mid x = (c\ y_1 \ldots\ y_n) \rightarrow (\widehat{\mathbb{L}}[(c\ y_1\ \ldots\ y_n)] \mid x = (c\ y_1\ \ldots\ y_n))$
	if c is a constructor, `return`, $\gg=$, `takeMVar`, `putMVar`, `newMVar`,
	or `future`
(mkbinds)	$\mathbb{L}[\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ s]$
	$\rightarrow \nu x_1 \ldots x_n.(\mathbb{L}[s] \mid x_1 = s_1 \mid\ \ldots\ \mid x_n = s_n)$
(seq)	$\mathbb{L}[(\texttt{seq}\ v\ s)] \rightarrow \mathbb{L}[s]$ if v is a functional value
(case)	$\mathbb{L}[\texttt{case}_T\ (c\ s_1 \ldots s_n)\ \texttt{of}\ \{\ldots((c\ y_1 \ldots y_n)\ \rightarrow\ s)\ldots\}]$
	$\rightarrow \nu y_1 \ldots y_n.(\mathbb{L}[s] \mid y_1 = s_1 \mid\ \ldots\ \mid y_n = s_n)$ if $n > 0$
(case)	$\mathbb{L}[\texttt{case}_T\ c\ \texttt{of}\ \{\ldots(c \rightarrow s)\ldots\}] \rightarrow \mathbb{L}[s]$

We now go through the rules for monadic computations. (lunit) implements the monadic sequencing operator $\gg=$, that is applicable if the monadic computation of the first argument is finished, hence of the form `return` s, and then the next computation (i.e. the second argument) of the sequence is started. A new filled MVar can be created using (nmvar), while a `takeMVar`-operation on a filled MVar can be performed using (tmvar) and a `putMVar`-operation on an empty MVar can be performed using (pmvar). Note that there are no rules for a `takeMVar`-operation on an empty MVar and a `putMVar`-operation on a filled MVar, this absence of such rules forces a thread to wait in such cases. (fork) creates a new thread where the return-value is a variable that identifies the thread. (unIO) binds the result of a monadic computation to a functional binding, thus the value of a concurrent future becomes accessible.

The rules for functional evaluations are explained as follows: (lbeta) implements the sharing-variant of a classical β-reduction. (cp), (cpcxa) and (cpcxb) inline a needed binding, where sharing of the call-by-need-evaluation is implemented by the new bindings. (mkbinds) moves a `letrec`-environment to the global bindings, where the ν-binders are used to restrict the scope of those bindings to the current thread. (seq) evaluates a `seq`-expression, where the first argument needs to be a value. The rule (case) evaluates `case`-expressions where the instantiation of pattern-variables for the appropriate `case`-alternative is implemented using bindings.

Note that if $P_1 \equiv \mathbb{D}[P_1']$, $P_2 \equiv \mathbb{D}[P_2']$ and $P_1' \to P_2'$ then $P_1 \to P_2$. Moreover we assume that all reduction rules obey the distinct variable convention, hence all bound expression variable names remain disjoint and differ from free variable names.

The *redex* of the reduction rules is defined as follows: For (lunit), (nmvar), (tmvar), (pmvar), (fork) it is the monadic expression in the context $\mathbb{M}$, for the rule (unIO) it is $y \gg\!= \texttt{return}\ s$, for (lbeta), (mkbinds), (seq), (case) it is the functional expression in the context $\mathbb{L}$, and for (cp), (cpcxa), (cpcxb) it is the variable x in the context $\widehat{\mathbb{L}}$.

Definition 2.22 (Standard Reduction)

1. Reduction step $s \xrightarrow{CHF^*} t$: Find the next reduction position of s using the process contexts of Definition 2.19 and then one applicable rule of Definition 2.21 yields t.
2. $s \xrightarrow{CHF^*,*} t$: s reduces to t with an arbitrary nonnegative number of reduction steps.
3. $s \xrightarrow{CHF^*,+} t$: s reduces to t with at least one reduction step.
4. $s \xrightarrow{CHF^*,k} t$: s reduces to t with exactly k reduction steps where $k \geq 0$.

Intuitively, two processes are seen as equal if their observable behavior is indistinguishable if both are plugged into any process context. We use the notions of may- and should-convergence as defined in (SS11) to observe the behavior of processes:

Definition 2.23 (May-Convergence and Should-Convergence)
Let P be a process.

1. P *may-converges* (written as $P{\downarrow}$), iff it is well-formed and reduces to a successful process, i.e.
$$P{\downarrow} \text{ iff } P \text{ is well-formed and } \exists P' : P \xrightarrow{CHF^*,*} P' \land P' \text{ is successful}$$
2. If $P{\downarrow}$ does not hold, then P *must-diverges* written as $P{\Uparrow}$.
3. P *should-converges* (written as $P{\Downarrow}$), iff it is well-formed and remains may-convergent after reductions, i.e.
$$P{\Downarrow} \text{ iff } P \text{ is well-formed and } \forall P' : P \xrightarrow{CHF^*,*} P' \implies P'{\downarrow}$$
4. If P is not should-convergent then we say P *may-diverges* written as $P{\uparrow}$.

We write $P{\downarrow}P'$ (or $P{\uparrow}P'$, resp.) if $P \xrightarrow{CHF^*,*} P'$ and P' is successful (or must-divergent, resp.).

Now we can define contextual equivalence as our notion of correctness:

Definition 2.24 (Contextual Equivalence and Program Transformation)

1. Contextual approximation $\leq_c$ is defined as $\leq_c := \ \leq_{\downarrow} \cap \leq_{\Downarrow}$.
2. Contextual may-equivalence $\sim_{\downarrow,c}$ is defined as $\sim_{\downarrow,c} := \leq_{\downarrow} \cap \geq_{\downarrow}$.
3. Contextual equivalence $\sim_c$ on processes is defined as $\sim_c := \leq_c \cap \geq_c$ where for $\xi \in \{\downarrow, \Downarrow\}$: $P_1 \leq_\xi P_2$ iff $\forall \mathbb{D} \in \mathit{PCtxt} : \mathbb{D}[P_1]\xi \Rightarrow \mathbb{D}[P_2]\xi$.

A *program transformation* P on processes is a binary relation on processes. It is *correct* iff $P \subseteq \sim_c$.

The calculus CHF as defined in (SS11) only has one rule (cpcx), while CHF* has the rules (cpcxa) and (cpcxb) instead. To show that both calculi are equivalent w.r.t. may- and should-convergence and also w.r.t. correctness of transformations, we need two additional program transformations given in Definition 2.25.

(cpx) copies variables and can be used to shorten chains. The first rule for (gc) removes bindings on process level, while the other two (gc)-rules remove bindings of `letrec`-environments.

Definition 2.25 (Program Transformations (cpx) **and** (gc)**)**

(cpx) $\mathbb{T}[x] \mid x = y \to \mathbb{T}[y] \mid x = y$
where we assume that it is closed w.r.t. $\mathbb{D}$-contexts and $\equiv$.

(gc) $\nu x_1, \ldots, x_n.(P \mid \texttt{Comp}(x_1) \mid \ldots \mid \texttt{Comp}(x_n)) \to P$
if for all $i \in \{1, \ldots, n\}$: $\texttt{Comp}(x_i)$ is a binding $x_i = s_i$, an MVar $x_i\ \mathbf{m}\ s_i$ or an empty MVar $x_i\ \mathbf{m}\ -$, and $x_i \notin FV(P)$.

(gc) $\mathbb{C}[\texttt{letrec}\ E\ \texttt{in}\ s] \to \mathbb{C}[s]$ if $FV(s) \cap LV(E) = \emptyset$

(gc) $\mathbb{C}[\texttt{letrec}\ E_1, E_2\ \texttt{in}\ s] \to \mathbb{C}[\texttt{letrec}\ E_2\ \texttt{in}\ s]$ if $FV(E_1, s) \cap LV(E_1) = \emptyset$

Theorem 2.1 (Equivalence of CHF and CHF*)
The calculi CHF and CHF* are equivalent w.r.t. may- and should-convergence and also w.r.t. correctness of transformations.

Proof
It suffices to show that $P \downarrow_{CHF^*} \iff P \downarrow_{CHF}$ and $P \Downarrow_{CHF^*} \iff P \Downarrow_{CHF}$ (or equivalently $P \uparrow_{CHF} \iff P \uparrow_{CHF^*}$) for all processes P. Thus we have to show four implications:

1. $P \downarrow_{CHF^*} \implies P \downarrow_{CHF}$: Let $P \xrightarrow{CHF^*,*} Q$, where Q is successful.

   Then the reduction can be translated into:

$$P \xrightarrow{CHF} \xleftrightarrow{cpx,*} \xleftrightarrow{gc,*} P_2 \xrightarrow{CHF} \xleftrightarrow{cpx,*} \xleftrightarrow{gc,*} \ldots Q$$

   Since the reductions (cpx) and (gc) are correct in CHF (SS11, SSS12), it is easy to show by induction on the number of $\xrightarrow{CHF}$-reductions, that $P \downarrow_{CHF}$.

2. $P \downarrow_{CHF} \implies P \downarrow_{CHF^*}$: Let $P \xrightarrow{CHF,*} Q$, where Q is successful.

   We transform it into a mixture of reductions and transformations in CHF*. All standard reductions are the same, with the exception of (cpcx) which is (cpcxa);(cpcxb) plus equivalences using $\xleftarrow{cpx,*}$ and $\xleftarrow{gc,*}$. The reduction

$$P_1[x] \mid x = c\, y_1 \ldots y_n \xrightarrow{cpcx} P_1[c\, z_1 \ldots z_n] \mid x = c\, z_1 \ldots z_n \mid z_1 = y_1 \mid \ldots \mid z_n = y_n$$

   is translated into

$$\begin{array}{ll} \xrightarrow{cpcxb} & P_1[c\, y_1 \ldots y_n] \mid x = c\, y_1 \ldots y_n \\ \xleftarrow{gc,*} & P_1[c\, y_1 \ldots y_n] \mid x = c\, y_1 \ldots y_n \mid z_1{=}y_1 \mid \ldots \mid z_n{=}y_n \\ \xleftarrow{cpx,*} & P_1[c\, z_1 \ldots z_n] \mid x = c\, z_1 \ldots z_n \mid z_1{=}y_1 \mid \ldots \mid z_n{=}y_n \end{array}$$

where we omit ν-binders. A step-wise transformation of the reduction sequence with the same intermediate processes is of the form

$$P \xrightarrow{CHF^*} \xleftarrow{gc,*} \xleftarrow{cpx,*} P_2 \xrightarrow{CHF^*} \xleftarrow{gc,*} \xleftarrow{cpx,*} \ldots Q$$

This reasoning is also applicable to (cpcxa)-reductions that only abstract some subexpressions. We modify the sequence into a CHF*-reduction sequence to a successful process, where scanning all possibilities of interference with the standard reductions of CHF* leads to the following diagrams:

$$\begin{tikzcd}
P \arrow[r, "cpx"] \arrow[d, "sr{,}a"'] & P' \arrow[d, dashed, "sr{,}a"] \\
P_1 \arrow[r, dashed, "\mathbb{T}{,}cpx"'] & P'_1
\end{tikzcd}
\quad
\begin{tikzcd}
P \arrow[rr, "cpx"] \arrow[d, "sr{,}cpcxb"'] & & P' \arrow[d, dashed, "sr{,}cpcxb"] \\
P_1 \arrow[r, dashed, "cpx"'] & \cdot \arrow[r, dashed, "\mathbb{T}{,}cpx"'] & P'_1
\end{tikzcd}
\quad
\begin{tikzcd}
P \arrow[r, "cpx"] \arrow[d, "sr{,}cp"'] & P' \arrow[dl, dashed, equal] \\
P_1 &
\end{tikzcd}
\quad
\begin{tikzcd}
P \arrow[r, "gc"] \arrow[d, "sr{,}a"'] & P' \arrow[d, dashed, "sr{,}a"] \\
P_1 \arrow[r, dashed, "gc"'] & P'_1
\end{tikzcd}$$

We use these diagrams to shift (gc) and (cpx) to the right, only over CHF*-reductions. We start with the rightmost of (cpx),(gc). This may increase the cpx-reductions or it may also remove a (cp)-reduction using the third diagram. Finally it leads to a sequence $P \xrightarrow{CHF^*,*} Q' (\xleftarrow{gc,*} \cdot \xleftarrow{cpx,*})^* Q$. This shifting terminates since the number of CHF*-reductions is not increased. It is easy to see that also Q' must be successful, since (cpx) and (gc) do no change this property. Hence we have shown that $P \downarrow_{CHF^*}$.

3. $P{\uparrow_{CHF^*}} \Longrightarrow P{\uparrow_{CHF}}$: Analogous to part 1, where Q is CHF*-must-diverging, which is CHF-must-diverging, since part 2 implies $Q{\Uparrow_{CHF}} \implies Q{\Uparrow_{CHF^*}}$.

4. $P{\uparrow_{CHF}} \Longrightarrow P{\uparrow_{CHF^*}}$: Let P be a process with a reduction sequence $P \xrightarrow{CHF,*} Q$, where $Q \Uparrow_{CHF}$.

   We use the same transformation as in part 2, which leads to a mixed reduction and transformation sequence $P\ (\xrightarrow{CHF^*} \cdot \xleftarrow{cpx,*} \cdot \xleftarrow{gc,*})^*\ Q$. The diagrams and the shifting process is the same as in part 2 and leads to a sequence $P \xrightarrow{CHF^*,*} Q'\ (\xleftarrow{cpx,*} \cdot \xleftarrow{gc,*})^*\ Q$. Now we have to argue that also Q' is CHF*-must-divergent. Since Q is CHF-must-divergent and since (cpx), (gc) are correct, we also obtain that Q' is CHF-must-divergent and part 1 implies $Q' \Uparrow_{CHF} \implies Q' \Uparrow_{CHF^*}$ and thus Q' is also CHF*-must-divergent.

Using Theorem 2.1 we can import the correctness-results for CHF* from (SS11, SSS12):

Proposition 2.2 (Correctness of Standard Reduction)
The transformations (lunit), (nmvar), (tmvar), (pmvar), (fork), (unIO), (lbeta), (cp), (cpcxa), (cpcxb), (mkbinds), (seq) and (case) are correct.

Nondeterminism is introduced in CHF* by the ability to concurrently access MVars from different threads. We want to give a short example:

Example 2.3 (Nondeterminism of CHF*-processes)
Consider the following process P:

$$y_1 \overset{\text{main}}{\Longleftarrow} \texttt{takeMVar}\ x \mid y_2 \Leftarrow \texttt{putMVar}\ x\ \texttt{True} \mid y_3 \Leftarrow \texttt{putMVar}\ x\ \texttt{False} \mid x\,\mathbf{m}\,-$$

Since the MVar x is empty and the main-thread wants to perform a take-operation on this MVar, the main-thread needs to wait. Then we have nondeterminism, since both concurrent threads are allowed to perform their put-operation on MVar x, resulting in either P_1 or P_2:

$$\begin{aligned}
P_1 &:= y_1 \overset{\text{main}}{\Longleftarrow} \texttt{takeMVar}\ x \mid y_2 \Leftarrow \texttt{return}\ () \mid y_3 \Leftarrow \texttt{putMVar}\ x\ \texttt{False} \mid x\,\mathbf{m}\,\texttt{True}\\
P_2 &:= y_1 \overset{\text{main}}{\Longleftarrow} \texttt{takeMVar}\ x \mid y_2 \Leftarrow \texttt{putMVar}\ x\ \texttt{True} \mid y_3 \Leftarrow \texttt{return}\ () \mid x\,\mathbf{m}\,\texttt{False}
\end{aligned}$$

For both P_1 and P_2 now the main-thread can perform its take-operation, resulting in the main-thread either returning `True` or `False` and therefore terminating the whole reduction sequence in both cases. A context can be easily given that shows that those two results are not contextual equivalent.

For better readability we sometimes use the do-notation as in Haskell, i.e. nested >>= operations are summarized as shown in the following example.

Example 2.4 (do-notation)
Consider the following CHF*-expression:

```
((newMVar True) >>=
   (λx.takeMVar x
           >>=
        (λy.future (putMVar x True)
             >>=
            (λy1.future (putMVar x False)
               >>=
                 (λy2.takeMVar x
                      >>=
                        (λr.return r)
                 )
            ))))
```

Using the do-notation and <- instead of ⇐, the above expression can be written as follows:

```
do x <- newMVar True
   takeMVar x
   y1 <- future (putMVar x True)
   y2 <- future (putMVar x False)
   r <- takeMVar x
   return r
```

Replacing `future` by `forkIO` yields a valid subprogram for Concurrent Haskell.

CHF* can be used to analyze Concurrent Haskell. Note that the future variables are lazy-evaluated at expression-level, while monadic operations are evaluated strictly.

In contrast to Concurrent Haskell, CHF* permits a lazy handling of infinite lists on process level using futures.

2.3 Datatype Declarations and Haskell-Notations

In Definition 2.1 and Definition 2.13 type-constructors are assumed to be defined. For completeness we give a grammar in Haskell-style that implements type-definitions for our let-polymorphism for both LRP and CHF* taken from (Dal16).

Definition 2.26 (Datatype Declarations)
Datatype declarations *DataDefs* are generated by the following grammar with $n \geq 1$ for every occurrence of n. ≀ is a placeholder for | to prevent a collision with the BNF-notation.

$$\begin{array}{lcl}
\mathit{DataDefs} & ::= & \mathit{DataDef}_1 \ \ldots \ \mathit{DataDef}_n \\
\mathit{DataDef} & ::= & \texttt{data}\ K\ a_1\ \ldots\ a_{ar(K)}\ =\ \mathit{constr}_1\ \wr\ \ldots\ \wr\ \mathit{constr}_n \\
\mathit{constr} & ::= & K\ \mathit{typeA}_1\ \ldots\ \mathit{typeA}_{ar(K)} \\
\mathit{typeA} & ::= & a \mid K \mid (K\ \mathit{typeA}_1\ \ldots\ \mathit{typeA}_n) \mid (\mathit{typeA}_1 \to \ldots \to \mathit{typeA}_n)
\end{array}$$

For the classical Haskell-notations `[]` and `(:)` for both LRP and CHF* the constructors `Nil` and `Cons` are used, but for better readability we often apply the Haskell-notation. Also `Zero` and `Succ` can be used to implement numbers using a Peano encoding (often written as usual numbers, e.g. 0 instead of `Zero`) and `True` and `False` are assumed to be defined as Boolean-constructors.

Haskell-expressions like `if-then-else`-expressions or function definitions can be implemented in LRP and on the functional level of CHF* as follows:

Haskell	LRP / CHF*
`if` b `then` s `else` t	`case` b `of` $\{(\texttt{True} \to s)\ (\texttt{False} \to t)\}$
$f\ x\ y = s$	`letrec` $f = \lambda x.\lambda y.s$ `in` $\ldots$
f `True False`	$((f\ \texttt{True})\ \texttt{False})$

2.4 Abstract Machines

In this chapter we consider abstract machines, that can be used to execute LRP- and CHF*-programs. This section is primarily based on (Sab12).

In the following subsections we present three abstract machines:

1. The abstract machine *M1* evaluates deterministic pure functional programs, is used for LRP and the deterministic part of CHF* and defined in Section 2.4.2.
2. *IOM1* is *M1* extended to handle monadic operations, is used for the deterministic and monadic part of CHF* and defined in Section 2.4.3.
3. *CIOM1* is *IOM1* extended by concurrency, used for the complete calculus CHF* and defined in Section 2.4.4.

For each abstract machine a simplified syntax is used, as defined in the next section.

2.4.1 Machine Language

The idea of simplified expressions is to simplify expressions by new bindings. We first give a definition for CHF* and then a compact definition for LRP.

Definition 2.27 (Simplified CHF*-Expression)
Let variables $x, x_i \in \mathit{Var}$. Every type constructor K has an arity $ar(K) \geq 0$ and a finite set D_K of data constructors $c_{K,i} \in D_K$ with an arity $ar(c_{K,i}) \geq 0$.
Simplified CHF*-expressions Expr_S and simplified monadic expressions MExpr_S are generated by the following grammar with $n \geq 1$:

$$\begin{array}{lcl} s, s_i \in \mathit{Expr}_S & ::= & me \mid x \mid (s\ x) \mid (\texttt{seq}\ s\ x) \mid (c_{K,i}\ x_1\ \dots\ x_{ar(c_{K,i})}) \mid (\lambda x.s) \\ & & \mid (\texttt{case}_K\ s\ \texttt{of}\ \{(\mathit{Pat}_{K,1} \to s_1)\ \dots\ (\mathit{Pat}_{K,|D_K|} \to s_{|D_K|})\}) \\ & & \mid (\texttt{letrec}\ x_1 = s_1, \dots, x_n = s_n\ \texttt{in}\ s) \\ \mathit{Pat}_{K,i} & ::= & (c_{K,i}\ x_1\ \dots\ x_{ar(c_{K,i})}) \\ me \in \mathit{MExpr}_S & ::= & \texttt{return}\ x \mid x_1 \gg\!= x_2 \mid \texttt{future}\ x \mid \texttt{takeMVar}\ x \mid \texttt{newMVar} \\ & & \mid \texttt{putMVar}\ x_1\ x_2 \end{array}$$

Simplified processes Proc_S are defined as Proc in Definition 2.13 where all expressions are simplified expressions and all MVars have only variables as content.

Definition 2.28 (Simplified LRP-Expression)
Simplified LRP-expressions are defined like Expr_S in Definition 2.27 but without monadic expressions.
Note that polymorphic abstractions and type applications are only needed for typing and are also omitted compared to the original LRP-syntax (see Definition 2.1).

Simplified expressions are also called *machine expressions*. Moreover we use Expr_S for both LRP and CHF* for better readability.

Now we give a transformation that translates a usual CHF*-program to a corresponding simplified program.

Definition 2.29 (Transformation from CHF*-Expressions to Machine Expressions)
$\sigma : \mathit{Proc} \to \mathit{Proc}_S$ simplifies CHF*-processes, where y and y_i are fresh variables.

- $\sigma(x) := x$
- $\sigma((\lambda x.s)) := (\lambda x.\sigma(s))$
- $\sigma((s\ t)) := \texttt{letrec}\ y = \sigma(t)\ \texttt{in}\ (\sigma(s)\ y)$
- $\sigma((\texttt{seq}\ s\ t)) := \texttt{letrec}\ y = \sigma(t)\ \texttt{in}\ \texttt{seq}\ \sigma(s)\ y$
- $\sigma((c\ s_1\ \dots\ s_{ar(c)})) := \texttt{letrec}\ y_1 = \sigma(s_1), \dots, y_{ar(c)} = \sigma(s_{ar(c)})\ \texttt{in}\ (c\ y_1\ \dots\ y_n)$
  if c is a constructor or a monadic operator
- $\sigma((\texttt{case}_K\ s\ \texttt{of}\ \{(\mathit{Pat}_{K,1} \to t_1)\ \dots\ (\mathit{Pat}_{K,|D_K|} \to t_{|D_K|})\}))$
  $:= (\texttt{case}_K\ \sigma(s)\ \texttt{of}\ \{(\mathit{Pat}_{K,1} \to \sigma(t_1))\ \dots\ (\mathit{Pat}_{K,|D_K|} \to \sigma(t_{|D_K|}))\})$
- $\sigma((\texttt{letrec}\ x_1 = s_1, \dots, x_n = s_n\ \texttt{in}\ t))$
  $:= (\texttt{letrec}\ x_1 = \sigma(s_1), \dots, x_n = \sigma(s_n)\ \texttt{in}\ \sigma(t))$
- $(P_1 \mid P_2) := (\sigma(P_1) \mid \sigma(P_2))$
- $\sigma(x \Leftarrow s) := x \Leftarrow \sigma(s)$
- $\sigma(\nu x.P) := \nu x.\sigma(P)$
- $\sigma(x\ \mathbf{m}\ s) := x\ \mathbf{m}\ y \mid y = \sigma(s)$
- $\sigma(x\ \mathbf{m}\ -) := x\ \mathbf{m}\ -$
- $\sigma(x = s) := x = \sigma(s)$

The transformation ψ for LRP is defined as follows and analogous to σ, but ψ also removes polymorphic abstractions, type abstractions and type annotations, since the used abstract machines does not carry type information. ψ is an adapted version of the transformation for LR in (SSS16b).

Definition 2.30 (Transformation from LRP-Expressions to Machine Expressions)
$\psi : \mathit{Expr} \to \mathit{Expr}_S$ simplifies LRP-expressions, where y and y_i are fresh variables.

- $\psi((\Lambda a_1.\Lambda a_2.\ldots\Lambda a_k.\lambda x :: \tau.s)) := (\lambda x.\psi(s))$
- $\psi(x :: \varphi) := x$
- $\psi((s\ \tau)) := \psi(s)$
- $\psi((s\ t)) := \texttt{letrec}\ y = \psi(t)\ \texttt{in}\ (\psi(s)\ y)$
- $\psi((\texttt{seq}\ s\ t)) := \texttt{letrec}\ y = \psi(t)\ \texttt{in}\ \texttt{seq}\ \psi(s)\ y$
- $\psi((c\ s_1\ \ldots\ s_{ar(c)})) := \texttt{letrec}\ y_1 = \psi(s_1), \ldots, y_{ar(c)} = \psi(s_{ar(c)})\ \texttt{in}\ (c\ y_1\ \ldots\ y_n)$
- $\psi((\texttt{case}_K\ s\ \texttt{of}\ \{(\mathit{Pat}_{K,1} \to t_1)\ \ldots\ (\mathit{Pat}_{K,|D_K|} \to t_{|D_K|})\}))$
  $:= (\texttt{case}_K\ \psi(s)\ \texttt{of}\ \{(\mathit{Pat}_{K,1} \to \psi(t_1))\ \ldots\ (\mathit{Pat}_{K,|D_K|} \to \psi(t_{|D_K|}))\})$
- $\psi((\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t))$
  $:= (\texttt{letrec}\ x_1 = \psi(s_1), \ldots, x_n = \psi(s_n)\ \texttt{in}\ \psi(t))$

Both transformations are correct:

Proposition 2.3 (Correctness of Transformation σ for CHF*)
$P \sim_c \sigma(P)$ holds for all processes $P \in \mathit{Proc}$.

Proof
(SS11) implies that the claim holds for CHF. Since Theorem 2.1 shows that CHF and CHF* are equivalent w.r.t. may- and should-convergence, the claim also holds for CHF*.

Proposition 2.4 (Correctness of Transformation ψ for LRP)
$e \sim_c \psi(e)$ holds for all expressions $e \in \mathit{Expr}$.

Proof
Follows directly from the results of (SSS16b).

Thus the machine languages, that are used by the following abstract machines, can be used for the analysis of LRP and CHF*.

2.4.2 Abstract Machine M1

In this section the abstract machine M1 is introduced and based on (SS15). This abstract machine is the abstract machine Mark 1 (Ses97) extended by the handling of `seq`- and `case`-expressions. Moreover monadic operators are treated as constructor applications and therefore the machine halts if it needs to evaluate monadic actions – this fits to the behavior of waiting threads later. Thus the abstract machine M1 is able to handle the complete calculus LRP and the deterministic part of CHF* using the corresponding machine language defined in Section 2.4.1.

We now start with the definition of M1-states:

Definition 2.31 (M1 State)
A state of M1 is a triple $\langle \mathcal{H} \mid s \mid \mathcal{S} \rangle$ consisting of the following components:
- The heap $\mathcal{H}$ is a mapping of variables to machine expressions, where the mapping can be written as $\{x_1 \mapsto e_1, \dots, x_n \mapsto e_n\}$. An empty heap is written as $\varnothing$ and $\mathcal{H}, x = s$ is the disjoint union of $\mathcal{H}$ and $\{x \mapsto s\}$.
- s is a machine expression, that is often called *control expression*, since it is the currently evaluated expression.
- The stack $\mathcal{S}$ contains entries of the form:
  - #app(x) to remember the second argument of an application,
  - #seq(x) to remember the second argument of a `seq`-expression,
  - #case$(alts)$ to remember the `case`-alternatives of a `case`-expression,
  - #upd(x) to remember a heap-binding that needs to be updated.

  The list notations is used for stacks, hence $[]$ denotes the empty stack, while $t : \mathcal{S}$ is a stack with top entry t and tail $\mathcal{S}$.

At the beginning of an execution heap and stack are empty:

Definition 2.32 (M1 Initial State)
Given a machine expression s, the corresponding M1 initial state is:

$$\langle \varnothing \mid s \mid [] \rangle$$

We also define the notion of values that is compatible to LRP and the deterministic part of CHF*.

Definition 2.33 (M1 Value)
A machine expression is an M1 value if it is an abstraction, constructor application or monadic expression.

A single transition from one state to the next one is defined by the following transition-rules, where at most one of these rules is applicable for the same state. Note that we assume that α-renaming is performed implicitly to preserve the distinct variable convention.

Definition 2.34 (M1 Transition Relation $\xrightarrow{M1}$)

(Unwind1)	$\langle \mathcal{H} \mid (s\ x) \mid \mathcal{S} \rangle \to \langle \mathcal{H} \mid s \mid \#\text{app}(x) : \mathcal{S} \rangle$
(Unwind2)	$\langle \mathcal{H} \mid (\texttt{seq}\ s\ x) \mid \mathcal{S} \rangle \to \langle \mathcal{H} \mid s \mid \#\text{seq}(x) : \mathcal{S} \rangle$
(Unwind3)	$\langle \mathcal{H} \mid \texttt{case}_K\ s\ \texttt{of}\ alts \mid \mathcal{S} \rangle \to \langle \mathcal{H} \mid s \mid \#\text{case}(alts) : \mathcal{S} \rangle$
(Lookup)	$\langle \mathcal{H}, x = s \mid x \mid \mathcal{S} \rangle \to \langle \mathcal{H} \mid s \mid \#\text{upd}(x) : \mathcal{S} \rangle$
(Letrec)	$\langle \mathcal{H} \mid \texttt{letrec}\ Env\ \texttt{in}\ s \mid \mathcal{S} \rangle \to \langle \mathcal{H}, Env \mid s \mid \mathcal{S} \rangle$
(Subst)	$\langle \mathcal{H} \mid (\lambda x.s) \mid \#\text{app}(y) : \mathcal{S} \rangle \to \langle \mathcal{H} \mid s[y/x] \mid \mathcal{S} \rangle$
(Branch)	$\langle \mathcal{H} \mid (c_{K,i}\ \vec{x}) \mid \#\text{case}(\dots\ ((c_{K,i}\ \vec{y}) \to t)\ \dots) : \mathcal{S} \rangle \to \langle \mathcal{H} \mid t[\vec{x}/\vec{y}] \mid \mathcal{S} \rangle$
(Seq)	$\langle \mathcal{H} \mid v \mid \#\text{seq}(x) : \mathcal{S} \rangle \to \langle \mathcal{H} \mid x \mid \mathcal{S} \rangle$ if v is a M1 value
(Update)	$\langle \mathcal{H} \mid v \mid \#\text{upd}(x) : \mathcal{S} \rangle \to \langle \mathcal{H}, x = v \mid v \mid \mathcal{S} \rangle$ if v is a M1 value
(Blackhole)	$\langle \mathcal{H} \mid x \mid \mathcal{S} \rangle \to \langle \mathcal{H} \mid x \mid \mathcal{S} \rangle$ if no binding for x exists in $\mathcal{H}$

(Unwind1), (Unwind2), (Unwind3) are used to handle the control-flow using the stack. (Lookup) moves a demanded heap binding into the scope of evaluation, while (Update) writes the result of a calculation back to the heap. (Subst) performs a β-reduction, (Seq) evaluates a `seq`-expression and (Branch) evaluates a `case`-expression. If a needed case-alternative is missing, i.e. a (Branch) should be performed but is not possible, then a runtime error occurs. (Blackhole) is an infinite loop and implements the waiting of a thread, until an MVar is accessible to perform the corresponding action.

A terminating program, starting with an initial state and a sequence of transition rules, yields a final state:

Definition 2.35 (M1 Final State)
Let v be a M1 value, then a *final state* is:

$$\langle \mathcal{H} \mid v \mid [] \rangle$$

We now give an example for an evaluation sequence on the M1:

Example 2.5 (Evaluation Sequence on M1)

$$\begin{array}{ll} \mathcal{H} & \emptyset \\ s & \texttt{letrec}\ x = ((\lambda k.k)\ \ y), y = \texttt{True}, z = (\texttt{newMVar}\ \ x)\ \ \texttt{in seq}\ \ x\ \ z \\ \mathcal{S} & [] \end{array}$$

$$\xrightarrow{\text{(Letrec)}}$$

$$\begin{array}{ll} \mathcal{H} & \{x = ((\lambda k.k)\ \ y),\ \ y = \texttt{True},\ \ z = (\texttt{newMVar}\ \ x)\} \\ s & \texttt{seq}\ \ x\ \ z \\ \mathcal{S} & [] \end{array}$$

$$\xrightarrow{\text{(Unwind2)}}$$

$$\begin{array}{ll} \mathcal{H} & \{x = ((\lambda k.k)\ \ y),\ \ y = \texttt{True},\ \ z = (\texttt{newMVar}\ \ x)\} \\ s & x \\ \mathcal{S} & [\texttt{\#seq}(z)] \end{array}$$

$$\xrightarrow{\text{(Lookup)}}$$

$$\begin{array}{ll} \mathcal{H} & \{y = \texttt{True},\ \ z = (\texttt{newMVar}\ \ x)\} \\ s & ((\lambda k.k)\ \ y) \\ \mathcal{S} & [\texttt{\#upd}(x), \texttt{\#seq}(z)] \end{array}$$

$$\xrightarrow{\text{(Unwind1)}}$$

$$\begin{array}{ll} \mathcal{H} & \{y = \texttt{True},\ \ z = (\texttt{newMVar}\ \ x)\} \\ s & (\lambda k.k) \\ \mathcal{S} & [\texttt{\#app}(y), \texttt{\#upd}(x), \texttt{\#seq}(z)] \end{array}$$

$$\xrightarrow{\text{(Subst)}}$$

$$\begin{array}{ll} \mathcal{H} & \{y = \texttt{True},\ \ z = (\texttt{newMVar}\ \ x)\} \\ s & y \\ \mathcal{S} & [\texttt{\#upd}(x), \texttt{\#seq}(z)] \end{array}$$

$$\xrightarrow{\text{(Lookup)}}$$

$$\begin{array}{ll} \mathcal{H} & \{z = (\texttt{newMVar}\ \ x)\} \\ s & \texttt{True} \\ \mathcal{S} & [\texttt{\#upd}(y), \texttt{\#upd}(x), \texttt{\#seq}(z)] \end{array}$$

$$\xrightarrow{\text{(Update)}}$$

$$\begin{array}{ll} \mathcal{H} & \{y = \texttt{True}, z = (\texttt{newMVar}\ \ x)\} \\ s & \texttt{True} \\ \mathcal{S} & [\texttt{\#upd}(x), \texttt{\#seq}(z)] \end{array}$$

$$\xrightarrow{\text{(Update)}}$$

$$\begin{array}{ll} \mathcal{H} & \{x = \texttt{True}, y = \texttt{True}, z = (\texttt{newMVar}\ \ x)\} \\ s & \texttt{True} \\ \mathcal{S} & [\texttt{\#seq}(z)] \end{array}$$

$$\xrightarrow{\text{(Seq), (Lookup), (Update)}}$$

$$\begin{array}{ll} \mathcal{H} & \{x = \texttt{True}, y = \texttt{True}\} \\ s & (\texttt{newMVar}\ \ x) \\ \mathcal{S} & [] \end{array}$$

The calculation stops since monadic actions are treated like constructor applications (see Definition 2.33).

The following measure is used to measure the runtime on the M1:

Definition 2.36 (Reduction Length Measures mln and mlnall)
Let s be a closed machine expression with $\langle \varnothing \mid s \mid [] \rangle \xrightarrow{n} Q$ where Q is a final state.

1. mln(s) is the sum of all (Subst)-, (Branch)- and (Seq)-steps in the sequence.
2. mlnall(s) is the number of all reductions steps in the sequence, hence n.

If no such sequence exists, then we have mln(s) = mlnall(s) = ∞. mln$(\cdot)$ is also used for reachable states Q.

As also showed in (SS15), the runtime measures rln and mln are compatible:

Proposition 2.5 (Compatibility of Runtime Measures rln and mln)
For any closed LRP-expression s, we have rln(s) = mln$(\psi(s))$

In summary the M1 can be used as abstract machine for LRP. (Update) does not allow to write a variable-to-variable-binding to the heap (the only difference to the M1 of (Sab12)) since LRP does not copy variables. In contrast to this, CHF* allows and needs the copying of variables. But we can use the M1 for the deterministic part of CHF* too, if we allow to write variable-to-variable-bindings to the heap in the (Update)-rule (compare Definition 2.34), if such a binding does not map to itself.

2.4.3 Abstract Machine IOM1

The abstract machine M1 of the last section treats monadic actions as constructor applications and as demonstrated in Example 2.5 monadic actions are completely ignored if we use the M1 for CHF*. In this section the abstract machine IOM1 is introduced, that is based on the M1 and additionally handles monadic actions. More concrete the handling of $\gg=$, `return`, `takeMVar`, `newMVar` and `putMVar` needs to be implemented. The IOM1 adds the monadic layer but is still not able to handle concurrent threads.

First we give a definition of IOM1-states, that is an extension of Definition 2.31, where the same notions are used (e.g. for the heap and stacks).

Definition 2.37 (IOM1 State)
A state of IOM1 is a tuple $\langle \mathcal{H} \mid \mathcal{M} \mid s \mid \mathcal{S} \mid \mathcal{I} \rangle$ consisting of the following components:

- The heap $\mathcal{H}$ is a mapping of variables to machine expressions.
- The set of MVars $\mathcal{M}$ is a mapping from names to machine expressions. This mapping can be seen as an additional heap, hence the same notions as for heaps can be used for MVars, but we often use the MVar-notion as before, e.g. $\{x \,\mathbf{m}\, -, y \,\mathbf{m}\, z\}$.
- s is a machine expression, that is often called *control expression*.
- The stack $\mathcal{S}$ contains entries of the form #app(x), #seq(x), #case$(alts)$ and #upd(x) as introduced in Definition 2.31.
- The IO-stack $\mathcal{I}$ contains entries of the form:
  - #take to remember a `takeMVar`-operation.
  - #put(x) to remember the content of a `putMVar`-operation.
  - #bind(x) to remember the second argument of a $\gg=$-expression.

The IO-stack is introduced to keep a clear separation between pure and monadic computations.

At the beginning of an execution heap, MVars, stack and IO-stack are empty:

Definition 2.38 (IOM1 Initial State)
Given a machine expression s, the corresponding IOM1 initial state is:

$$\langle \emptyset \mid \emptyset \mid s \mid [] \mid [] \rangle$$

A single transition from one state to the next one is defined by the following transition-rules, where at most one of these rules is applicable for the same state.

Definition 2.39 (IOM1 Transition Relation $\xrightarrow{\text{IOM1}}$)

(Unwind1)	$\langle \mathcal{H} \mid \mathcal{M} \mid (s\ x) \mid \mathcal{S} \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid s \mid \#\text{app}(x) : \mathcal{S} \mid \mathcal{I} \rangle$
(Unwind2)	$\langle \mathcal{H} \mid \mathcal{M} \mid (\texttt{seq}\ s\ x) \mid \mathcal{S} \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid s \mid \#\text{seq}(x) : \mathcal{S} \mid \mathcal{I} \rangle$
(Unwind3)	$\langle \mathcal{H} \mid \mathcal{M} \mid \texttt{case}_K\ s\ \texttt{of}\ alts \mid \mathcal{S} \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid s \mid \#\text{case}(alts) : \mathcal{S} \mid \mathcal{I} \rangle$
(Unwind4)	$\langle \mathcal{H} \mid \mathcal{M} \mid \texttt{takeMVar}\ x \mid [] \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid x \mid [] \mid \#\text{take} : \mathcal{I} \rangle$
(Unwind5)	$\langle \mathcal{H} \mid \mathcal{M} \mid \texttt{putMVar}\ x\ y \mid [] \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid x \mid [] \mid \#\text{put}(y) : \mathcal{I} \rangle$
(Unwind6)	$\langle \mathcal{H} \mid \mathcal{M} \mid x \gg= y \mid [] \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid x \mid [] \mid \#\text{bind}(y) : \mathcal{I} \rangle$
(Lookup)	$\langle \mathcal{H}, x = s \mid \mathcal{M} \mid x \mid \mathcal{S} \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid s \mid \#\text{upd}(x) : \mathcal{S} \mid \mathcal{I} \rangle$
(Letrec)	$\langle \mathcal{H} \mid \mathcal{M} \mid \texttt{letrec}\ Env\ \texttt{in}\ s \mid \mathcal{S} \mid \mathcal{I} \rangle \to \langle \mathcal{H}, Env \mid \mathcal{M} \mid s \mid \mathcal{S} \mid \mathcal{I} \rangle$
(Subst)	$\langle \mathcal{H} \mid \mathcal{M} \mid \lambda x.s \mid \#\text{app}(y) : \mathcal{S} \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid s[y/x] \mid \mathcal{S} \mid \mathcal{I} \rangle$
(Branch)	$\langle \mathcal{H} \mid \mathcal{M} \mid c_{K,i}\ \overrightarrow{x} \mid \#\text{case}(\dots\ ((c_{K,i}\ \overrightarrow{y}) \to t)\ \dots) : \mathcal{S} \mid \mathcal{I} \rangle$ $\to \langle \mathcal{H} \mid \mathcal{M} \mid t[\overrightarrow{x}/\overrightarrow{y}] \mid \mathcal{S} \mid \mathcal{I} \rangle$
(Seq)	$\langle \mathcal{H} \mid \mathcal{M} \mid v \mid \#\text{seq}(x) : \mathcal{S} \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid x \mid \mathcal{S} \mid \mathcal{I} \rangle$ if v is a M1 value
(Update)	$\langle \mathcal{H} \mid \mathcal{M} \mid v \mid \#\text{upd}(x) : \mathcal{S} \mid \mathcal{I} \rangle \to \langle \mathcal{H}, x = v \mid \mathcal{M} \mid v \mid \mathcal{S} \mid \mathcal{I} \rangle$ if v is a M1 value or a variable with $v \neq x$
(NewMVar)	$\langle \mathcal{H} \mid \mathcal{M} \mid \texttt{newMVar}\ x \mid [] \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M}, y\ \mathbf{m}\ x \mid \texttt{return}\ y \mid [] \mid \mathcal{I} \rangle$ where y is a fresh variable
(TakeMVar)	$\langle \mathcal{H} \mid \mathcal{M}, x\ \mathbf{m}\ y \mid x \mid [] \mid \#\text{take} : \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M}, x\ \mathbf{m}\ - \mid \texttt{return}\ y \mid [] \mid \mathcal{I} \rangle$
(PutMVar)	$\langle \mathcal{H} \mid \mathcal{M}, x\ \mathbf{m}\ - \mid x \mid [] \mid \#\text{put}(y) : \mathcal{I} \rangle$ $\to \langle \mathcal{H} \mid \mathcal{M}, x\ \mathbf{m}\ y \mid \texttt{return}\ () \mid [] \mid \mathcal{I} \rangle$
(LUnit)	$\langle \mathcal{H} \mid \mathcal{M} \mid \texttt{return}\ x \mid [] \mid \#\text{bind}(y) : \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid (y\ x) \mid [] \mid \mathcal{I} \rangle$
(Blackhole)	$\langle \mathcal{H} \mid \mathcal{M} \mid x \mid \mathcal{S} \mid \mathcal{I} \rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid x \mid \mathcal{S} \mid \mathcal{I} \rangle$ if no binding for x exists in $\mathcal{H}$ or $\mathcal{M}$

All rules from the M1 (see Definition 2.34) are adapted to the definition of IOM1-states. In contrast to the M1, (Update) now allows variables to be written back to the heap since CHF* allows variables to be copied while LRP does not.

Compared to the M1 the three rules (Unwind4), (Unwind5) and (Unwind6) are straightforward extensions for the control flow, required by the monadic layer.

(NewMVar) creates a new MVar with given content and returns a reference to this MVar in form of the variable, that is the name of the MVar. (TakeMVar) empties a filled MVar and returns the content of the MVar, while (PutMVar) fills an empty MVar with the given content. Note that there are no rules to apply (TakeMVar) on an empty

MVar or (PutMVar) on a filled MVar, hence the abstract machine gets stuck and in this way needs to wait until the preconditions are met to perform the corresponding operation. The evaluation of the $\gg=$-operator starts with the evaluation of the first argument using (Unwind6) and then is completed using (LUnit).

We also assume that α-renaming is performed implicitly to preserve the distinct variable convention.

A terminating program, starting with an initial state and a sequence of transition rules, yields a final state:

Definition 2.40 (IOM1 Final State)
An IOM1-state is called *final state* if it is of the form $\langle \mathcal{H} \mid \mathcal{M} \mid \texttt{return}\ x \mid [] \mid [] \rangle$ or $\langle \mathcal{H} \mid \mathcal{M} \mid v \mid [] \mid [] \rangle$, where v is an abstraction, constructor application or the name of an MVar.

We now give an example for an evaluation sequence on the IOM1:

Example 2.6 (Evaluation Sequence on IOM1)

$\mathcal{H}$	$\varnothing$
$\mathcal{M}$	$\varnothing$
s	`letrec` x = `True`, y = (`newMVar` x), z = (λk.`takeMVar` k) `in` $y \gg= z$
$\mathcal{S}$	[]
$\mathcal{I}$	[]

$$\xrightarrow{\text{(Letrec)}}$$

$\mathcal{H}$	{x = `True`, y = (`newMVar` x), z = (λk.`takeMVar` k)}
$\mathcal{M}$	$\varnothing$
s	$y \gg= z$
$\mathcal{S}$	[]
$\mathcal{I}$	[]

$$\xrightarrow{\text{(Unwind6)}}$$

$\mathcal{H}$	{x = `True`, y = (`newMVar` x), z = (λk.`takeMVar` k)}
$\mathcal{M}$	$\varnothing$
s	y
$\mathcal{S}$	[]
$\mathcal{I}$	[`#bind`(z)]

$$\xrightarrow{\text{(Lookup)}}$$

$\mathcal{H}$	{x = `True`, z = (λk.`takeMVar` k)}
$\mathcal{M}$	$\varnothing$
s	(`newMVar` x)
$\mathcal{S}$	[`#upd`(y)]
$\mathcal{I}$	[`#bind`(z)]

$$\xrightarrow{\text{(Update)}}$$

$\mathcal{H}$	{x = `True`, y = (`newMVar` x), z = (λk.`takeMVar` k)}
$\mathcal{M}$	$\varnothing$
s	(`newMVar` x)
$\mathcal{S}$	[]
$\mathcal{I}$	[`#bind`(z)]

$$\xrightarrow{\text{(NewMVar)}}$$

$\mathcal{H}$	{x = `True`, y = (`newMVar` x), z = (λk.`takeMVar` k)}
$\mathcal{M}$	$\{l\,\mathbf{m}\,x\}$
s	(`return` l)
$\mathcal{S}$	[]
$\mathcal{I}$	[`#bind`(z)]

$$\xrightarrow{\text{(LUnit)}}$$

$\mathcal{H}$	$\{x = \texttt{True},\ \ y = (\texttt{newMVar}\ \ x),\ \ z = (\lambda k.\texttt{takeMVar}\ \ k)\}$
$\mathcal{M}$	$\{l\,\mathbf{m}\,x\}$
s	$(z\ \ l)$
$\mathcal{S}$	$[]$
$\mathcal{I}$	$[]$

$$\xrightarrow{\text{(Unwind1)}}$$

$\mathcal{H}$	$\{x = \texttt{True},\ \ y = (\texttt{newMVar}\ \ x),\ \ z = (\lambda k.\texttt{takeMVar}\ \ k)\}$
$\mathcal{M}$	$\{l\,\mathbf{m}\,x\}$
s	z
$\mathcal{S}$	$[\texttt{\#app}(l)]$
$\mathcal{I}$	$[]$

$$\xrightarrow{\text{(Lookup)}}$$

$\mathcal{H}$	$\{x = \texttt{True},\ \ y = (\texttt{newMVar}\ \ x)\}$
$\mathcal{M}$	$\{l\,\mathbf{m}\,x\}$
s	$(\lambda k.\texttt{takeMVar}\ \ k)$
$\mathcal{S}$	$[\texttt{\#upd}(z), \texttt{\#app}(l)]$
$\mathcal{I}$	$[]$

$$\xrightarrow{\text{(Update)}}$$

$\mathcal{H}$	$\{x = \texttt{True},\ \ y = (\texttt{newMVar}\ \ x),\ \ z = (\lambda k.\texttt{takeMVar}\ \ k)\}$
$\mathcal{M}$	$\{l\,\mathbf{m}\,x\}$
s	$(\lambda k.\texttt{takeMVar}\ \ k)$
$\mathcal{S}$	$[\texttt{\#app}(l)]$
$\mathcal{I}$	$[]$

$$\xrightarrow{\text{(Subst)}}$$

$\mathcal{H}$	$\{x = \texttt{True},\ \ y = (\texttt{newMVar}\ \ x),\ \ z = (\lambda k.\texttt{takeMVar}\ \ k)\}$
$\mathcal{M}$	$\{l\,\mathbf{m}\,x\}$
s	$(\texttt{takeMVar}\ \ l)$
$\mathcal{S}$	$[]$
$\mathcal{I}$	$[]$

$$\xrightarrow{\text{(TakeMVar)}}$$

$\mathcal{H}$	$\{x = \texttt{True},\ \ y = (\texttt{newMVar}\ \ x),\ \ z = (\lambda k.\texttt{takeMVar}\ \ k)\}$
$\mathcal{M}$	$\{l\,\mathbf{m}\,-\}$
s	$(\texttt{return}\ \ x)$
$\mathcal{S}$	$[]$
$\mathcal{I}$	$[]$

In this section the machine M1 was included implicitly, since in this way it is easier to take care of the difference of copying variables between LRP and CHF*. Thus the M1 should be used for LRP only and the IOM1 should be used for CHF* only. In summary the IOM1 can be used as abstract machine for CHF*-programs that do not use concurrent threads.

2.4.4 Abstract Machine CIOM1

In this section the abstract machine CIOM1 is introduced, that uses the IOM1 to evaluate a single thread, but is also able to handle the creation and termination of threads. Thus the CIOM1 is able to execute CHF*-programs without restrictions.

First of all we define the notion of a thread:

Definition 2.41 (CIOM1 Thread)
A *thread* (also called *future*) of the CIOM1 is a tuple $(x, s, \mathcal{S}, \mathcal{I})$ consisting of the following components:

- x is a variable that is used as name of the thread.
- s is the current control expression.
- $\mathcal{S}$ is a stack.
- $\mathcal{I}$ is an IO-stack.

Each thread has a unique name, control expression, local stack and IO-stack. But the heap and MVars are shared between the threads. This yields the following definition of CIOM1-states:

Definition 2.42 (CIOM1 State)
A state of CIOM1 is a triple $\langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T} \rangle$ consisting of the following components:

- The heap $\mathcal{H}$ is a mapping of variables to machine expressions that is shared by all threads.
- The set of MVars $\mathcal{M}$ is a mapping from names to machine expressions that is shared by all threads.
- The set $\mathcal{T}$ contains all threads, where all threads have pairwise distinct names and exactly one thread has the name `main` (often called main-thread).

The calculation starts with the initial expression that is evaluated by the main-thread – further threads may be started later.

Definition 2.43 (CIOM1 Initial State)
Given a machine expression s, the corresponding CIOM1 initial state is:

$$\langle \varnothing \mid \varnothing \mid \{\ (\texttt{main}, s, [], [])\ \} \rangle$$

A single transition from one state to the next one is defined by the following transition-rules:

Definition 2.44 (CIOM1 Transition Relation $\xrightarrow{\text{CIOM1}}$)

(UnIO) $\langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T}, (x, \texttt{return}\ y, [], []) \rangle \to \langle \mathcal{H}, x = y \mid \mathcal{M} \mid \mathcal{T} \rangle$
if thread x is not the main-thread.

(Fork) $\langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T}, (x, \texttt{future}\ y, [], \mathcal{I}) \rangle$
$\to \langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T}, (x, \texttt{return}\ z, [], \mathcal{I}), (z, y, [], []) \rangle$
where z is a fresh variable.

(IOM1) $\langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T}, (x, s, \mathcal{S}, \mathcal{I}) \rangle \to \langle \mathcal{H}' \mid \mathcal{M}' \mid \mathcal{T}, (x, s', \mathcal{S}', \mathcal{I}') \rangle$
if $\langle \mathcal{H} \mid \mathcal{M} \mid s \mid \mathcal{S} \mid \mathcal{I} \rangle \xrightarrow{\text{IOM1}} \langle \mathcal{H}' \mid \mathcal{M}' \mid s' \mid \mathcal{S}' \mid \mathcal{I}' \rangle$ on machine IOM1.
This rule is only used if (UnIO) or (Fork) is not applicable for thread x.

In each transition-step one of all threads is chosen nondeterministically with the requirement that the thread is able to proceed. (UnIO) stores the result of a calculation of a thread on the heap and hereby makes the result accessible for the other threads. Threads needing such a result are forced to wait until it becomes available on the heap. (Fork) spawns a new thread. If none of these cases apply, then the machine IOM1 is used to continue the evaluation of the thread.

If the main-thread terminates, then all other threads also terminate:

Definition 2.45 (CIOM1 Final State)
A CIOM1-state is called *final state* if the thread with name `main` is of the form $(\texttt{main}, \texttt{return}\ x, [], [])$ or $(\texttt{main}, v, [], [])$, where v is an abstraction, a constructor application or the name of an MVar.

The used notion for convergence and divergence for the CIOM1 is analogous to the notion of CHF* in Definition 2.23: For may-convergence we write $s{\downarrow}_{\text{CIOM1}}$, for must-divergence $s{\Uparrow}_{\text{CIOM1}}$, for should-convergence $s{\Downarrow}_{\text{CIOM1}}$ and for may-divergence $s{\uparrow}_{\text{CIOM1}}$.

As also showed in (Sab12) the machine CIOM1 preserves the semantics:

Proposition 2.6 (Semantical Compatibility of CHF* and CIOM1)
For every expression $s :: \texttt{IO}\ \tau$ the equivalences $s{\downarrow} \iff s{\downarrow}_{\text{CIOM1}}$ and $s{\Downarrow} \iff s{\Downarrow}_{\text{CIOM1}}$ hold.

In summary the abstract machine CIOM1 can be used for CHF*. Since this machine is based on the machine IOM1 it should not be used for the evaluation of LRP-programs.

2.5 Proof Techniques

For many kinds of proofs we use forking and commuting diagrams to analyze all cases. The complete technique is defined in detail in (SSS08b). This sections aims to give an intuitive presentation of the technique.

If we add an additional reduction rule T to a calculus like LRP or CHF*, then for example we can try to show that the new rule is correct or improves the runtime or space behavior. This requires to consider normal order reduction sequences of the corresponding calculus, in the following written as $\xrightarrow{no}$, and to check all cases where T can be applied. In contrast to all transformations applied during the normal order reduction sequence T is not part of $\xrightarrow{no}$. Intuitively we look for all overlaps between the $\xrightarrow{no}$ and T, thus we look for forkings and commutings: A *forking* is the pattern $\xleftarrow{no,a} \cdot \xrightarrow{T}$, where a is an arbitrary normal order reduction rule and a *commuting* is the pattern $\xrightarrow{T} \cdot \xrightarrow{no,a}$. Forkings and commutings can be represented by forking diagrams as on the left and commuting diagrams as on the right:

$$\begin{tikzcd}
\cdot \arrow[r, "T"] \arrow[d, "no{,}a"'] & \cdot \arrow[d, dashed, "no{,}a"] \\
\cdot \arrow[r, dashed, "T"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "T"] \arrow[d, dashed, "no{,}a"'] & \cdot \arrow[d, "no{,}a"] \\
\cdot \arrow[r, dashed, "T"'] & \cdot
\end{tikzcd}$$

The solid arrows of a forking diagram are given transformations and the dashed arrows are existentially quantified transformations that close the forking. The same applies to commuting diagrams. $\rightsquigarrow$ can be used to shorten diagrams, e.g. $\xleftarrow{no,a} \cdot \xrightarrow{T} \rightsquigarrow \xrightarrow{T} \cdot \xleftarrow{no,a}$ for the left one.

To show a property of a program transformation, we usually use induction on the normal order reduction sequence and consider all diagrams in the induction step. For this approach different measures are used and depend on the calculus and considered transformations.

2.6 Function Definitions

In this section we give definitions of frequently used functions.

The identity function `id` is an abstraction and `comp` composes two functions:

$$\begin{aligned}
\texttt{id} &= \lambda x.x \\
\texttt{comp} &= \lambda f,g.(\lambda x.f\ (g\ x))
\end{aligned}$$

The classical Boolean-functions operators for negation, and, or and exclusive-or:

$$\begin{aligned}
\texttt{neg} &= \lambda x.\texttt{case}\ x\ \texttt{of}\ \{(\texttt{True} \to \texttt{False})\ (\texttt{False} \to \texttt{True})\} \\
\texttt{and} &= \lambda x,y.\texttt{case}\ x\ \texttt{of}\ \{(\texttt{True} \to y)\ (\texttt{False} \to \texttt{False})\} \\
\texttt{or} &= \lambda x,y.\texttt{case}\ x\ \texttt{of}\ \{(\texttt{True} \to \texttt{True})\ (\texttt{False} \to y)\} \\
\texttt{xor} &= \lambda x,y.\texttt{case}\ x\ \texttt{of}\ \{(\texttt{True} \to \texttt{case}\ y\ \texttt{of}\ \{(\texttt{True} \to \texttt{False})\ (\texttt{False} \to \texttt{True})\}) \\
&\qquad\qquad\qquad\qquad (\texttt{False} \to y)\}
\end{aligned}$$

The `fold`-functions from Haskell are defined as follows, where `foldl` is a left-fold, `foldl'` a more strict variant of `foldl` and `foldr` a right-fold:

$$\begin{aligned}
\texttt{foldl} &= \lambda f,z,xs.\texttt{case}\ xs\ \texttt{of}\ \{([\,] \to z)\ ((y:ys) \to \texttt{foldl}\ f\ (f\ z\ y)\ ys)\} \\
\texttt{foldl'} &= \lambda f,z,xs.\texttt{case}\ xs\ \texttt{of}\ \{([\,] \to z)\ ((y:ys) \to \texttt{letrec}\ w = (f\ z\ y)\ \texttt{in} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{seq}\ w\ (\texttt{foldl'}\ f\ w\ ys))\} \\
\texttt{foldr} &= \lambda f,z,xs.\texttt{case}\ xs\ \texttt{of}\ \{([\,] \to z)\ ((y:ys) \to f\ y\ (\texttt{foldr}\ f\ z\ ys))\}
\end{aligned}$$

Functions for lists, i.e. the usual null-, head-, tail-, map- and filter-functions, append `++` to concatenate two lists, `concat` for concatenation of lists, `replicate` to generate a list of given length k as Peano number containing k-times the second argument, `take` to return a specified part from the front of the list, a naive implementation of `reverse` to reverse the order of the elements of a list, `repeat` for an infinite list of the specified argument, `last` to get the last element of a list, `all` that returns `True` iff all elements fulfill the specified predicate and `allTrue` as direct implementation of `all (==True)`:

$$\begin{aligned}
\texttt{null} &= \lambda xxs.\texttt{case}\ xxs\ \texttt{of}\ \{([\,] \to \texttt{True})\ ((x:xs) \to \texttt{False})\} \\
\texttt{head} &= \lambda xxs.\texttt{case}\ xxs\ \texttt{of}\ \{((x:xs) \to x)\} \\
\texttt{tail} &= \lambda xxs.\texttt{case}\ xxs\ \texttt{of}\ \{([\,] \to \bot)\ ((x:xs) \to xs)\} \\
\texttt{map} &= \lambda f,xxs.\texttt{case}\ xxs\ \texttt{of}\ \{([\,] \to [\,])\ ((x:xs) \to ((f\ x):(\texttt{map}\ f\ xs)))\} \\
\texttt{filter} &= \lambda p,xxs.\texttt{case}\ xxs\ \texttt{of}\ \{([\,] \to [\,]) \\
&\qquad\qquad\qquad ((x:xs) \to \texttt{case}\ (p\ x)\ \texttt{of}\ \{ \\
&\qquad\qquad\qquad\qquad\qquad (\texttt{True} \to (x:(\texttt{filter}\ p\ xs))) \\
&\qquad\qquad\qquad\qquad\qquad (\texttt{False} \to \texttt{filter}\ p\ xs)\})\} \\
\texttt{++} &= \lambda xxs,yys.\texttt{case}\ xxs\ \texttt{of}\ \{([\,] \to yys)\ ((x:xs) \to (x:(\texttt{++}\ xs\ yys)))\} \\
\texttt{concat} &= \lambda xs.(\texttt{foldr}\ (\lambda x,y.\texttt{foldr}\ (\lambda z,zs.(z:zs))\ y\ x)\ [\,]\ xs) \\
\texttt{replicate} &= \lambda n,x.\texttt{case}\ n\ \texttt{of}\ \{(\texttt{Zero} \to [\,]) \\
&\qquad\qquad\qquad ((\texttt{Succ}\ m) \to (x:(\texttt{replicate}\ m\ x)))\}
\end{aligned}$$

```
take     = λc, xxs.case c of {
                     (Zero → [])
                     ((Succ x) → case xxs of {
                                   ([] → [])
                                   ((y : ys) → (y : (take x ys)))})}
reverse  = λxs.case xs of {([] → [])
                           ((y : ys) → reverse ys ++ [y])}
repeat   = λx.(x : (repeat x))
last     = λxxs.case xxs of {([] → ⊥)
                             ((x : xs) → case xs of {
                                           ([] → x)
                                           ((y : ys) → (last xs))})}
all      = λp, xxs.case xxs of {([] → True)
                                ((x : xs) → case (p x) of {
                                              (True → all p xs)
                                              (False → False)})}
allTrue  = λxxs.case xxs of {([] → True)
                             ((x : xs) → case x of {
                                           (True → allTrue xs)
                                           (False → False)})}
```

Operations on Peano numbers, i.e. addition, lower and greater:

```
padd     = λn, m.case m of {(Zero → n)
                            ((Succ x) → (Succ (padd n x)))}
plower   = λn, m.case n of {
                   (Zero → case m of {(Zero → False) ((Succ x) → True)})
                   ((Succ n′) → case m of {
                                  (Zero → False)
                                  ((Succ m′) → plower n′ m′)})}
pgreater = λn, m.case n of {
                   (Zero → False)
                   ((Succ n′) → case m of {
                                  (Zero → True)
                                  ((Succ m′) → pgreater n′ m′)})}
```

fBySnd takes a list of pairs and returns the pair with either the minimal (if plower is used as function) or maximal (if pgreater is used) second component.

```
fBySnd  = λf, xxs.case (null xxs) of {
                    (True → ⊥)
                    (False → fBySnd' (tail xxs) f (head xxs))}
fBySnd' = λxxs, f, p.case xxs of {
                       ([] → p)
                       ((x : xs) →
                          case x of {
```

```
((n, pt) →
   case p of {
      ((cN, cPT) →
         case (f pt cPT) of {
            (True → fBySnd' xs f (n, pt))
            (False → fBySnd' xs f (cN, cPT))})})
})}
```

3

Space Optimizations in LRP

In earlier work (e.g. (SSS17, SSS15a, SS15, SSS16b, SSS15b)) time improvements are studied. A time improvement is a transformation that improves the runtime in any case, hence it can be applied to any applicable subexpression if the semantics is not changed. For a better intuition we give a simplified definition of time improvements based on (SSS16b): Let s and t be LRP-expressions with the same type, then s *time-improves* t ($s \preceq t$) if $s \sim_c t$ and for all contexts $\mathbb{C}[\cdot]$ s.t. $\mathbb{C}[s]$ and $\mathbb{C}[t]$ are closed, $\text{rln}(s) \leq \text{rln}(t)$ holds.

The goal of this chapter is to analyze the space behavior of lazy-evaluating functional programs using a similar approach. As calculus LRP (see Section 2.1) is used.

Compared to the notion of the time improvements the behavior of evaluation w.r.t. space usage is often quite opaque and might lead to a much higher space usage as needed. (GS99, GS01, BR00) observed that semantically correct modifications of the sequence of evaluation (for example due to strictness information) may have a dramatic effect on space usage, where an example is (`head` xs) `eqBool` (`last` xs) vs. (`last` xs) `eqBool` (`head` xs) where xs is bound to an expression that generates a long list of Booleans, `head` returns the first element of a list, `last` the last element and `eqBool` tests whether two Boolean-values are equal.

In the area of call-by-need evaluation there are several analyses of space consumption by Gustavsson and Sands (GS99, GS01, Gus01). Their notion of space improvement is comparable to the notion we introduce in this chapter, but they use an untyped language and we will see further differences later.

This chapter is primarily based on (SSD18) and (SSD17).

3.1 LRP with Garbage Collection

In this section the calculus LRPgc is defined, that is LRP with an eager garbage collection: After each normal order reduction step garbage collection is applied. If garbage is considered at space measurement, then this might have a great impact on the space measurement and the analyses are not stable w.r.t. garbage collection. The goal is to build a foundation for the analysis of space and not for the analysis of garbage.

As garbage collection only the top `letrec` is considered, that intuitively corresponds to the heap of the abstract machine M1 (compare Section 2.4.2).

Definition 3.1 (Garbage Collection Transformation Rules for LRPgc)

(gc1) $\texttt{letrec}\ \{x_i = s_i\}_{i=1}^n, E\ \texttt{in}\ t \to \texttt{letrec}\ E\ \texttt{in}\ t$ if $\forall i : x_i \notin FV(t, E)$ and $n > 0$

(gc2) $\texttt{letrec}\ \{x_1 = s_1, \dots, x_n = s_n\}\ \texttt{in}\ t \to t$ if $\forall i : x_i \notin FV(t)$

(gc) is the union of (gc1) and (gc2).

LRPgc is the same as LRP but applies (gc) after each normal order reduction step:

Definition 3.2 (LRPgc)
Let the calculus LRPgc be LRP with a slightly changed normal order reduction:
A *normal-order-gc reduction step* $s \xrightarrow{LRPgc} t$ is defined by two cases:

1. If a (gc)-transformation is applicable to s in the empty context, i.e. $s \xrightarrow{gc} t$, then $s \xrightarrow{LRPgc} t$, where the maximum of bindings is removed.
2. If (1) is not applicable and $s \xrightarrow{LRP} t$, then $s \xrightarrow{LRPgc} t$.

A sequence of LRPgc-reduction steps is called a *normal-order-gc reduction sequence* or *LRPgc-reduction sequence.* A WHNF without the possibility of a $\xrightarrow{LRPgc,gc}$-reduction step is called *LRPgc-WHNF.* If the LRPgc-reduction sequence of an expression s halts with an LRPgc-WHNF, then s *converges* w.r.t. LRPgc, denoted as $s{\downarrow}_{LRPgc}$, or $s{\downarrow}$ if the calculus is clear from the context.
The calculus LRgc is defined as LRPgc without types.
The definition of context classes like $\mathbb{R}$, $\mathbb{T}$ and $\mathbb{S}$ are directly transferred to LRPgc.

We now give a formal definition of the syntactical size of LRP-expressions:

Definition 3.3 (LRP Syntactical Size)
The syntactical size $\texttt{synsize}(s)$ of an LRP-expression s is defined as:

$$\begin{array}{ll}
\texttt{synsize}(x) & = 1\\
\texttt{synsize}(s\ t) & = 1 + \texttt{synsize}(s) + \texttt{synsize}(t)\\
\texttt{synsize}(\lambda x.s) & = 2 + \texttt{synsize}(s)\\
\texttt{synsize}(\texttt{case}\ s\ \texttt{of}\ \{\texttt{alt}_1 \dots \texttt{alt}_n\}) & = 1 + \texttt{synsize}(s) + \sum_{i=1}^n \texttt{synsize}(\texttt{alt}_i)\\
\texttt{synsize}((c\ x_1 \dots x_n)\ \to\ s) & = 1 + n + \texttt{synsize}(s)\\
\texttt{synsize}(c\ s_1 \dots s_n) & = 1 + \sum \texttt{synsize}(s_i)\\
\texttt{synsize}(\texttt{seq}\ s\ t) & = 1 + \texttt{synsize}(s) + \texttt{synsize}(t)\\
\texttt{synsize}(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^n\ \texttt{in}\ s) & = 1 + n + \texttt{synsize}(s) + \sum \texttt{synsize}(s_i)
\end{array}$$

The following definition was originally given for LR in (SSS08b) and can be directly transferred to LRP:

Definition 3.4 (Measure μ_{lll})
The measure $\mu_{lll}(s)$ for an LRP-expression s is defined as follows:
$\mu_{lll}(s)$ is a pair $(\mu_{lll,1}(s), \mu_{lll,2}(s))$, ordered lexicographically. The measure $\mu_{lll,1}(s)$ is the number of `letrec`-subexpressions in s, and $\mu_{lll,2}(s)$ is the sum of $\texttt{lrdepth}(\mathbb{C})$ for all `letrec`-subexpressions r with $s = \mathbb{C}[r]$, where `lrdepth` is defined as follows, where $\mathbb{C}_{(1)}$ is a context of hole depth 1:

$$\begin{array}{ll}
\texttt{lrdepth}([\cdot]) & = 0\\
\texttt{lrdepth}(\mathbb{C}_{(1)}[\mathbb{C}'[]]) & = \begin{cases} 1 + \texttt{lrdepth}(\mathbb{C}'[]) & \text{if } \mathbb{C}_{(1)} \text{ is not a } \texttt{letrec}\\ \texttt{lrdepth}(\mathbb{C}'[]) & \text{if } \mathbb{C}_{(1)} \text{ is a } \texttt{letrec}\end{cases}
\end{array}$$

The measure is used for inductive proofs on the steps of a reduction sequence as well as the following lemma:

Lemma 3.1 (Inequations for μ_{lll})

1. If $s \xrightarrow{lll} t$, then $\mu_{lll}(s) > \mu_{lll}(t)$.
2. If $s \xrightarrow{\mathbb{T},gc} t$, then $\mu_{lll}(s) \geq \mu_{lll}(t)$.
3. If $s \xrightarrow{\mathbb{T},seq} t$, then $\mu_{lll}(s) \geq \mu_{lll}(t)$.

Proof
(1) is proved in (SSS08b) and it is obvious for (2) and (3).

Now we show that the addition of garbage collection does neither change convergence nor correctness. We start with the forking diagrams between $(\mathbb{T}, gc)$-reductions and LRPgc-reductions.

Lemma 3.2 (Complete Set of Forking Diagrams for $(\mathbb{T}, gc)$-reductions)
The forking diagrams between $(\mathbb{T}, gc)$- and LRPgc-reductions are the following:

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T},gc"] \arrow[d, "LRPgc,a"'] & t \arrow[d, dashed, "LRPgc,a"] \\
s' \arrow[r, dashed, "\mathbb{T},gc,*"'] & t'
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{T},gc"] \arrow[d, "LRPgc,gc"'] & t \\
s' \arrow[ur, dashed, "\mathbb{T},gc,*"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[d, "LRPgc,gc"'] \arrow[d, dashed, bend left, "\mathbb{T},gc"] \\
s'
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{T},gc"] \arrow[dd, "LRPgc,a"'] & t \arrow[d, dashed, "LRPgc,gc"] \\
 & t_1 \arrow[d, dashed, "LRPgc,a"] \\
s' \arrow[r, dashed, "\mathbb{T},gc,*"'] & t'
\end{tikzcd}$$

$a \in \{gc, lll\}$ $\qquad\qquad$ $a \notin \{gc, lll\}$

Proof
The diagrams for $a \neq gc$ can be derived from the appendix in (SSS08b).

For $s \xrightarrow{LRPgc,gc} s'$, the bottom reduction may consist of 0, 1 or 2 (gc)-reductions. A typical example for the latter case is:

$$\begin{tikzcd}
\begin{array}{l}\texttt{letrec } x_1 = \texttt{True}, x_2 = \texttt{True in} \\ \quad(\texttt{letrec } y_1 = x_2 \texttt{ in False})\end{array} \arrow[r, "\mathbb{T},gc"] \arrow[d, "LRPgc,gc"'] & \begin{array}{l}\texttt{letrec } x_1 = \texttt{True}, x_2 = \texttt{True in} \\ \qquad\texttt{False}\end{array} \arrow[d, dashed, "LRPgc,gc"] \\
\begin{array}{l}\texttt{letrec } x_2 = \texttt{True in} \\ (\texttt{letrec } y_1 = x_2 \texttt{ in False})\end{array} \arrow[r, dashed, "\mathbb{T},gc,*"'] & \texttt{False}
\end{tikzcd}$$

Lemma 3.3 (Equality of Number of *LCSC*-steps after $(\mathbb{T}, gc)$)
Let $s \xrightarrow{\mathbb{T},gc} t$ with $s{\downarrow}_{LRP}$. If s has an LRP-reduction with n *LCSC*-reductions, then this also holds for t.

Proof
Let $s \xrightarrow{LRPgc,*} s'$ be an LRPgc-reduction such that s' is a WHNF and let $s \xrightarrow{\mathbb{T},gc} t$. We show that claim by induction on (i) the number of *LCSC*-reductions of s to a WHNF,

(ii) on the μ_{lll}-measure and (iii) on the syntactical size. The diagrams referenced in the following are the diagrams of Lemma 3.2.

- If $s \xrightarrow{LRPgc,a} s'$ where a is a *LCSC*-reduction, then we can apply the induction hypothesis to s', and the reduction sequence to t' using diagram 1 and 4. We then obtain that the number of *LCSC*-reductions of s' is the same as for s.
- If $s \xrightarrow{LRPgc,lll} s'$, then we can apply the induction hypothesis to s' and thus also to t' using diagrams 1 and 2. We then obtain that the number of *LCSC*-reductions is the same for s and t.
- If $s \xrightarrow{LRPgc,gc} s'$, then we can apply the induction hypothesis to s'. For diagram 1, the reasoning is as above. For diagram 2, we can apply the induction hypothesis to s' and t and obtain the claim. For diagram 3, the claim is obvious.

Finally we can show the convergence-equivalence between LRP and LRPgc:

Proposition 3.1 (Convergence-Equivalence between LRP and LRPgc)
The calculus LRP is convergence-equivalent to LRPgc. I.e. for all expressions s: $s{\downarrow}_{LRP} \iff s{\downarrow}_{LRPgc}$.
Also contextual equivalence and preorder for LRP coincides with the corresponding notions in LRPgc.

Proof
Let $s{\downarrow}_{LRPgc}$. Then it is sufficient to argue as for LR by induction on the number of normal-order reductions:

If s is a WHNF, then the claim is trivial. If $s \xrightarrow{LRPgc,a} s'$, then there are two cases:

1. $s \xrightarrow{LRPgc,a} s'$ with $a \neq gc$, then using the induction hypothesis we obtain $s{\downarrow}_{LRP}$.
2. $s \xrightarrow{gc} s'$. By induction we obtain $s'{\downarrow}_{LRP}$ and since (gc) is correct in LR, we also obtain $s{\downarrow}_{LRP}$.

For the reversed implication, let s be an expression with $s{\downarrow}_{LRP}$. We have to construct a LRPgc-reduction to a WHNF. The induction is on the number of *LCSC*-reductions of s, then on the measure μ_{lll} and then on the size of s seen as syntax tree. The decrease of the measure μ_{lll} for (lll)- and for (gc)-reductions is shown in Lemma 3.1.

1. s is a WHNF. Then there is a sequence $s \xrightarrow{LRPgc,gc,*} s'$, where s' is a LRPgc-WHNF.
2. $s \xrightarrow{LRP} s'$ and the reduction is also a LRPgc-reduction. Then obviously $s'{\downarrow}_{LRP}$. If the reduction is a *LCSC*-reduction, then we can apply the induction hypothesis to s'. If the reduction is a (lll)-reduction, then the expression s' is smaller w.r.t. μ_{lll} and we can apply the induction hypothesis to s'.
3. The third case is $s \xrightarrow{LRPgc,gc} s'$. Then either $\mu_{lll}(s) > \mu_{lll}(s')$ or $\mu_{lll}(s) = \mu_{lll}(s')$. Moreover the syntactical size is strictly decreased and $s'{\downarrow}_{LRP}$ by Lemma 3.1 and 3.3. Then we can apply the induction hypothesis and obtain a LRPgc-reduction of s to a WHNF.

3.2 Definition of Space Improvement and Equivalence

To be able to analyze the space behavior of programs, the space measurement needs to be defined. First of all we give a definition of the size of expressions, that is a little bit weaker than the syntactical size (compare Definition 3.3):

Definition 3.5 (LRP Size of Expressions size)
The size $\mathtt{size}(s)$ of an LRP-expression s is defined as:

$$\begin{array}{ll}
\mathtt{size}(x) & = 0 \\
\mathtt{size}(s\ t) & = 1 + \mathtt{size}(s) + \mathtt{size}(t) \\
\mathtt{size}(\lambda x.s) & = 1 + \mathtt{size}(s) \\
\mathtt{size}(\mathtt{case}\ s\ \mathtt{of}\ \{\mathtt{alt}_1 \dots \mathtt{alt}_n\}) & = 1 + \mathtt{size}(s) + \sum_{i=1}^{n} \mathtt{size}(\mathtt{alt}_i) \\
\mathtt{size}((c\ x_1 \dots x_n)\ \to\ s) & = 1 + \mathtt{size}(s) \\
\mathtt{size}(c\ s_1 \dots s_n) & = 1 + \sum \mathtt{size}(s_i) \\
\mathtt{size}(\mathtt{seq}\ s\ t) & = 1 + \mathtt{size}(s) + \mathtt{size}(t) \\
\mathtt{size}(\mathtt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \mathtt{in}\ s) & = \mathtt{size}(s) + \sum \mathtt{size}(s_i)
\end{array}$$

Type annotations and type expressions are ignored by the `size`-measure, since it corresponds to the number of nodes of the whole program seen as sharing graph.

Variables are not counted by the measure, since variable-to-variable bindings like $x = y$ are indirections that are not created by abstract machines – the abstract machine M1 directly substitutes and never creates such an indirection (see Definition 2.34). Also an abstract machine uses a heap that can be seen as a global `letrec`-expression, hence all let-reduction-rules do not change the size and therefore the `letrec`-expression itself is not counted by `size`.

The sizes `size` and `synsize` only differ by a constant factor:

Proposition 3.2 (Deviation between synsize and size)
Let s be an LRP-expression. If s does not permit a garbage collection of any binding and there are no $x = y$-bindings, then $\mathtt{synsize}(s) \leq (\mathit{maxarity} + 1) \cdot \mathtt{size}(s)$ and $\mathtt{size}(s) \leq \mathtt{synsize}(s)$, where *maxarity* is the maximum of 2 and the maximal arity of constructor symbols in the language.

Proof
It is sufficient to check every subexpression using an inductive argument.

Based on the measure `size` for expressions, a space measure for evaluation sequences is defined as follows:

Definition 3.6 (LRP Space Measure *spmax*)
The space measure $\mathit{spmax}(s)$ for the reduction sequence of a closed expression s is the maximum of those $\mathtt{size}(s_i)$, where $s_i \xrightarrow{LRPgc} s_{i+1}$ is not a (gc), and where the reduction sequence is $s = s_0 \xrightarrow{LRPgc} s_1 \xrightarrow{LRPgc} \dots \xrightarrow{LRPgc} s_n$, and s_n is a WHNF. If $s\Uparrow$, then $\mathit{spmax}(s)$ is defined as ∞.
For a (partial) reduction sequence $\mathit{Red} = s_1 \to \dots \to s_n$ we define:

$$\mathit{spmax}(\mathit{Red}) = \max\{size(s_i) \mid s_i \to s_{i+1} \text{ is not a (gc) and also } s_n \text{ is not LRPgc-reducible with a (gc)-reduction}\}$$

Counting space only if there is no (LRPgc,gc)-reduction step possible is consistent with the definition in (GS01). It also has the effect of avoiding certain small and short peaks in the space usage. The advantage is a better correspondence with the abstract machine and it leads to comprehensive results.

Now we can give a definition of space improvements and equivalences.

Definition 3.7 (LRP Space Improvement and Space Equivalence)
Let s and t be two LRP-expressions with $s \sim_c t$ and $s\downarrow$.

- s is a *space improvement* of t, $s \leq_{spmax} t$, if for all contexts $\mathbb{C}$:
  If $\mathbb{C}[s]$, $\mathbb{C}[t]$ are closed then $spmax(\mathbb{C}[s]) \leq spmax(\mathbb{C}[t])$.
- s is *space-equivalent* to t, $s \sim_{spmax} t$, if for all contexts $\mathbb{C}$:
  If $\mathbb{C}[s]$, $\mathbb{C}[t]$ are closed then $spmax(\mathbb{C}[s]) = spmax(\mathbb{C}[t])$.

A transformation $\xrightarrow{trans}$ is called a *space improvement* (*space equivalence*) if $t \xrightarrow{trans} s$ implies that s is a space improvement of (space-equivalent to, respectively) t. Often we also say *maximal space improvement* and *maximal space equivalence* to express the used measure *spmax*. Also we often say s *space-improves* t.
We write $s \leq_{X,spmax} t$ ($s \sim_{X,spmax} t$) for a context class X, to denote that the definition is as above but restricted to context class X. E.g. for $s \leq_{\mathbb{R},spmax} t$, if $\mathbb{R}[s]$ and $\mathbb{R}[t]$ are closed, we require $spmax(\mathbb{R}[s]) \leq spmax(\mathbb{R}[t])$ for all reduction contexts $\mathbb{R}$.
Note that $t \geq_{spmax} s$ is sometimes used instead of $s \leq_{spmax} t$.

$\leq_{spmax}$ is a precongruence, i.e. it is transitive and $s \leq_{spmax} t$ implies $C[s] \leq_{spmax} C[t]$. $\sim_{spmax}$ is a congruence.

For each subexpression where the pattern of a space-improving transformation applies, then the transformation can be used without increasing the space consumption. Thus such a transformation can be applied to any proper subprogram without the risk of increasing the space consumption. The same holds for space equivalences where the space consumption is also not allowed to be decreased.

The following lemma shows a correspondence between space improvement and `size` of the associated expressions.

Lemma 3.4 (Space Improvement Property Implies Less or Equal `size`)
If $s \leq_{spmax} t$ for two LRP-expressions s and t, then $\texttt{size}(s) \leq \texttt{size}(t)$.

Proof
The context $\lambda x.[\cdot]$ for a fresh variable x enforces $\texttt{size}(s) \leq \texttt{size}(t)$.

We also consider useful program-transformations that are runtime optimizations, but may increase the space usage during runtime and distinguish acceptable and bad behavior w.r.t. space usage. Transformations that applied in reduction contexts lead to a space increase of at most a fixed (additive) constant are considered as controllable and safe, whereas the case that after the transformation the space increase may exceed any constant (depending on the usage of the expressions), is considered uncontrollable and we say it is a space leak:

Definition 3.8 (Space-Leak)
Let T be a transformation and let $s \xrightarrow{T} t$ be an instance with LRP-expressions s and t.

1. We say that $s \xrightarrow{T} t$ is *space-safe up to the constant* c, if for all reduction contexts $\mathbb{R}$: *spmax*$(\mathbb{R}[t]) \leq c +$ *spmax*$(\mathbb{R}[s])$.
2. If for some c, (1) holds for all instances $s \xrightarrow{T} t$, then we say T is *space-safe up to the constant* c.
3. The transformation $s \xrightarrow{T} t$ is a *space leak*, if and only if for every real number b, there is a reduction context $\mathbb{R}$, such that *spmax*$(\mathbb{R}[t]) \geq b +$ *spmax*$(\mathbb{R}[s])$.
4. If there is one instance $s \xrightarrow{T} t$ that is a space leak, then we also say T is a *space leak.*

This definition is a first criterion for a classification of transformations. Definition 3.8 for a classification of transformations makes sense insofar as space improvements are not space leaks and space leaks cannot be space improvements.

3.3 Context Lemmas for Space Improvement and Equivalence

The space improvement definition considers all contexts. Therefore we use a so-called *context lemma* that reduces the amount of cases that need to be considered in the case analyses of interferences between normal order reductions and transformations.

Between LRPgc and LRgc we have the same relation as between LRP and LR, hence we only use LRgc in the following and can transfer the results to LRPgc.

First we analyze the impact of a garbage collection in the empty context on the space consumption:

Lemma 3.5 (Impact of (gc) **on** ***spmax*** **in Empty Context)**
Let s be an expression with $s \xrightarrow{[\cdot],gc} s'$, i.e. it is a gc-reduction step in the empty context. Then *spmax*$(s) =$ *spmax*(s').

Proof
There are two cases:

1. If $s \xrightarrow{gc} s'$ is $s \xrightarrow{LRgc,gc} s'$, then the claim holds by applying *spmax*.
2. In the other case the (gc)-transformation is a (gc1)-reduction and the following diagram holds:

$$\begin{tikzcd} s \arrow[r, "gc"] \arrow[d, dashed, "LRgc{,}gc1"'] & s' \arrow[dl, dashed, "LRgc{,}gc"] \\ s_1 & \end{tikzcd}$$

Hence *spmax*$(s) =$ *spmax*$(s_1) =$ *spmax*(s').

The context lemma for space improvements is as follows, where we use Lemma 3.5 in the proof:

Lemma 3.6 (Context Lemma for Maximal Space Improvements)
If $\texttt{size}(s) \leq \texttt{size}(t)$, $FV(s) \subseteq FV(t)$ and $s \leq_{\mathbb{R},spmax} t$, then $s \leq_{spmax} t$.

Proof
Let M be a multi-context. We prove the more general claim that if $M[s_1,\ldots,s_n]$ and $M[t_1,\ldots,t_n]$ are closed and $M[s_1,\ldots,s_n]{\downarrow}$ and for all i: $\texttt{size}(s_i) \leq \texttt{size}(t_i)$, $FV(s_i) \subseteq FV(t_i)$, $s_i \leq_{R,spmax} t_i$, then $spmax(M[s_1,\ldots,s_n]) \leq spmax(M[t_1,\ldots,t_n])$.

By the assumption that $s_i \sim_c t_i$, we have $M[s_1,\ldots,s_n] \sim_c M[t_1,\ldots,t_n]$ and thus $M[s_1,\ldots,s_n]{\downarrow} \iff M[t_1,\ldots,t_n]{\downarrow}$. The induction proof is first on the number of LRPgc-reduction steps of $M[t_1,\ldots,t_n]$ and as a second parameter on the number of holes of M. We distinguish the following cases:

1. The LRPgc-reduction step of $M[t_1,\ldots,t_n]$ is a (gc).

   If M is the empty context, then we can apply the assumption $s_1 \leq_{R,spmax} t_1$, which shows $spmax(s_1) \leq spmax(t_1)$.

   Now we can assume that M is not empty, hence it is a context starting with a `letrec` and in $M[t_1,\ldots,t_n]$ the reduction (gc) removes a subset of the bindings in the top letrec, resulting in $M'[t'_1,\ldots,t'_k]$. Since $FV(s_i) \subseteq FV(t_i)$, the same set of bindings in the top `letrec` can be removed in $M[s_1,\ldots,s_n]$ by (gc) resulting in $M'[s'_1,\ldots,s'_k]$, where the pairs (s'_i,t'_i) are renamed versions of pairs (s_j,t_j).

   If the reduction is a (gc2) or a (gc1) with $M[s_1,\ldots,s_n] \xrightarrow{LRgc,gc1} M'[s'_1,\ldots,s'_k]$, then by induction we obtain $spmax(M'[s'_1,\ldots,s'_k]) \leq spmax(M'[t'_1,\ldots,t'_k])$. Since $spmax$ is not changed by (gc)-reduction, this shows the claim.

   However, a (gc1)-step, that is not a LRPgc-reduction step, does not remove the maximal set of removable bindings in $M[s_1,\ldots,s_n]$. By induction we obtain $spmax(M'[s'_1,\ldots,s'_k]) \leq spmax(M'[t'_1,\ldots,t'_k])$. We use Lemma 3.5, which shows $spmax(M'[s'_1,\ldots,s'_k]) = spmax(M[s_1,\ldots,s_n])$ and also the equation $spmax(M'[t'_1,\ldots,t'_k]) = spmax(M[t_1,\ldots,t_n])$ holds, and thus the claim is shown.

2. If no hole of M is in a reduction context and the reduction step is not a (gc), then there are two cases:

   – $M[t_1,\ldots,t_n]$ is a WHNF. Then also $M[s_1,\ldots,s_n]$ is a WHNF and by the assumption, we have $\texttt{size}(M[s_1,\ldots,s_n]) \leq \texttt{size}(M[t_1,\ldots,t_n])$.

   – We have $M[t_1,\ldots,t_n] \xrightarrow{LRgc,a} M'[t'_1,\ldots,t'_{n'}]$ and $M[s_1,\ldots,s_n] \xrightarrow{LRgc,a} M'[s'_1,\ldots,s'_{n'}]$ with $a \neq gc$ and the pairs (s'_i,t'_i) are renamed versions of pairs (s_j,t_j). This shows $spmax(M'[s'_1,\ldots,s'_{n'}]) \leq spmax(M'[t'_1,\ldots,t'_{n'}])$ by induction.
     By assumption, the inequation $\texttt{size}(M[s_1,\ldots,s_n]) \leq \texttt{size}(M[t_1,\ldots,t_n])$ holds, hence $spmax(M[s_1,\ldots,s_n]) \leq spmax(M[t_1,\ldots,t_n])$ is obtained by computing the maximum.

3. Some t_j in $M[t_1,\ldots,t_n]$ is in a reduction position and there is no LRPgc-(gc)-reduction of $M[t_1,\ldots,t_n]$. Then there is one hole, say i, of M that is in a reduction position. With $M' = M[\cdot,\ldots,\cdot,t_i,\cdot,\ldots,\cdot]$, we can apply the induction hypothesis, since the number of holes of M' is strictly smaller than the number of holes of M and the number of normal-order-gc reduction steps of $M[t_1,\ldots,t_n]$ is the same as of $M'[t_1,\ldots,t_{i-1},t_{i+1},\ldots,t_n]$ and obtain:

   $$\mathit{spmax}(M[s_1,\ldots,s_{i-1},t_i,s_{i+1},\ldots,s_n]) \leq \mathit{spmax}(M[t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_n])$$

   Also since $M[s_1,\ldots,s_{i-1},\cdot,s_{i+1},\ldots,s_n]$ is a reduction context, the assumption yields:

   $$\mathit{spmax}(M[s_1,\ldots,s_{i-1},s_i,s_{i+1},\ldots,s_n]) \leq \mathit{spmax}(M[s_1,\ldots,s_{i-1},t_i,s_{i+1},\ldots,s_n])$$

   Hence $\mathit{spmax}(M[s_1,\ldots,s_n]) \leq \mathit{spmax}(M[t_1,\ldots,t_n])$.

The requirements of the context lemma are necessary:

First we consider the requirement $FV(s) \subseteq FV(t)$: Let s = `letrec` $y = x$ `in True` and let t = `letrec` y = `True in True`. Then $s \sim_c t$, since s and t are both contextually equivalent to `True`, using garbage collection. Also $\texttt{size}(s) \leq \texttt{size}(t)$. But s is not a maximal space improvement of t:

Let $\mathbb{C}$ be the context `letrec` $x = s_1, z = s_2$ `in seq` z (`seq` (c $[\cdot]$) z), where s_1 and s_2 are closed expressions such that $\texttt{size}(s_1) \geq 2$ and the evaluation of s_2 produces a WHNF $s_{2,\mathit{WHNF}}$ of size at least $1 + \texttt{size}(s_1) + \texttt{size}(s_2)$. This is easy to construct using recursive list functions. Then the reduction sequence of $\mathbb{C}[s]$ reaches the size maximum after s_2 is reduced to WHNF due to the first `seq`, which is $1 + \texttt{size}(s_1) + \texttt{size}(s_{2,\mathit{WHNF}}) + 3 + \texttt{size}(s)$. The reduction sequence of $\mathbb{C}[t]$ first removes s_1 and then reaches the same maximum as s, which is $1 + \texttt{size}(s_{2,\mathit{WHNF}}) + 3 + \texttt{size}(t)$. Thus $\mathit{spmax}(\mathbb{C}[s]) - \mathit{spmax}([t]) = \texttt{size}(s_1) + \texttt{size}(s) - \texttt{size}(t) = \texttt{size}(s_1) - 1 > 0$.

We have to show that $\mathit{spmax}(\mathbb{R}[s]) \leq \mathit{spmax}(\mathbb{R}[t])$ holds for all reduction contexts $\mathbb{R}$:

Reducing $\mathbb{R}[s]$ will first shift (perhaps in several steps) the binding $y = x$ to the top letrec and then remove it (perhaps with other bindings) with (gc). The same for $\mathbb{R}[t]$. After this removal, the expressions are the same. Hence $\mathit{spmax}(\mathbb{R}[s]) \leq \mathit{spmax}(\mathbb{R}[t])$. This shows that if $FV(s) \subseteq FV(t)$ is violated, then the context lemma does not hold in general. Note that this example also shows, that for arbitrary expressions s and t with $s \sim_c t$ and $s\downarrow$, the relation $s \leq_{\mathbb{R},\mathit{spmax}} t$ does not imply $FV(s) \subseteq FV(t)$.

The requirement $\texttt{size}(s) \leq \texttt{size}(t)$ is also needed: Let t be a small expression that generates a large WHNF and let s be `seq True` t. Then $\texttt{size}(s) > \texttt{size}(t)$. Lemma 3.4 shows by contradiction that s cannot be a space improvement of t. For all reduction contexts $\mathbb{R}$, the first non-(gc) reduction will join the reduction sequences of $\mathbb{R}[s]$ and $\mathbb{R}[t]$. Since the WHNF of s is large, we obtain $\mathit{spmax}(\mathbb{R}[s]) = \mathit{spmax}(\mathbb{R}[t])$, since the difference in size of 1 between s and t, is too small compared with the size of the WHNF. This implies $s \leq_{R,\mathit{spmax}} t$, but s is not a maximal space improvement of t. Thus the condition $\texttt{size}(s) \leq \texttt{size}(t)$ is necessary in Lemma 3.6.

For cases that do not change maximal space consumption, we adapt Lemma 3.6 as follows:

Lemma 3.7 (Context Lemma for Maximal Space Equivalences)
If $\mathtt{size}(s) = \mathtt{size}(t)$, $FV(s) = FV(t)$ and $s \sim_{\mathbb{R},spmax} t$, then $s \sim_{spmax} t$.

Proof
Follows by applying Lemma 3.6 in both directions.

The context lemmas also obviously hold for stronger context classes:

Proposition 3.3 (Applicability of Context Lemmas for Stronger Context Classes)
The context lemmas Lemma 3.6 and Lemma 3.7 also hold for all context classes that contain reduction contexts.
Especially both lemmas also hold for top and surface contexts.

The context lemmas for maximal space improvement and maximal space equivalence in the polymorphic variant LRPgc cannot be derived from the context lemmas in LRgc. However, it is easy to see that the reasoning in the proofs of the context lemmas is completely analogous and so we obtain:

Proposition 3.4 (Context Lemmas for LRgc also hold for LRPgc)
The context lemmas for maximal space improvement (Lemma 3.6) and maximal space equivalence (Lemma 3.7) also hold in LRPgc.

3.4 Space-Safety of Transformations

In this section the space behavior of many transformations is analyzed. Several extra transformations are needed in the following:

Definition 3.9 (Extra LRPTransformation Rules)

(cpx-in) $(\mathtt{letrec}\ x = y, E\ \mathtt{in}\ \mathbb{C}[x]) \to (\mathtt{letrec}\ x = y, E\ \mathtt{in}\ \mathbb{C}[y])$
where y is a variable and $x \neq y$

(cpx-e) $(\mathtt{letrec}\ x = y, z = \mathbb{C}[x], E\ \mathtt{in}\ t) \to (\mathtt{letrec}\ x = y, z = \mathbb{C}[y], E\ \mathtt{in}\ t)$
where y is a variable and $x \neq y$

(cpcx-in) $(\mathtt{letrec}\ x = c\ \vec{t}, E\ \mathtt{in}\ \mathbb{C}[x])$
$\to (\mathtt{letrec}\ x = c\ \vec{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, E\ \mathtt{in}\ \mathbb{C}[c\ \vec{y}])$

(cpcx-e) $(\mathtt{letrec}\ x = c\ \vec{t}, z = \mathbb{C}[x], E\ \mathtt{in}\ t)$
$\to (\mathtt{letrec}\ x = c\ \vec{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, z = \mathbb{C}[c\ \vec{y}], E\ \mathtt{in}\ t)$

(abs) $(\mathtt{letrec}\ x = c\ \vec{t}, E\ \ \mathtt{in}\ s) \to (\mathtt{letrec}\ x = c\ \vec{x}, \{x_i = t_i\}_{i=1}^{ar(c)}, E\ \ \mathtt{in}\ s)$
where $ar(c) \geq 1$

(abse) $(c\ \vec{t}) \to (\mathtt{letrec}\ \{x_i = t_i\}_{i=1}^{ar(c)}\ \mathtt{in}\ c\ \vec{x})$ where $ar(c) \geq 1$

(xch) $(\mathtt{letrec}\ x = t, y = x, E\ \mathtt{in}\ r) \to (\mathtt{letrec}\ y = t, x = y, E\ \mathtt{in}\ r)$

(ucp1) $(\mathtt{letrec}\ E, x = t\ \mathtt{in}\ \mathbb{S}[x]) \to (\mathtt{letrec}\ E\ \mathtt{in}\ \mathbb{S}[t])$

(ucp2) $(\mathtt{letrec}\ E, x = t, y = \mathbb{S}[x]\ \mathtt{in}\ r) \to (\mathtt{letrec}\ E, y = \mathbb{S}[t]\ \mathtt{in}\ r)$

(ucp3) $(\mathtt{letrec}\ x = t\ \mathtt{in}\ \mathbb{S}[x]) \to \mathbb{S}[t]$
where in the three (ucp)-rules, x has at most one occurrence in $\mathbb{S}[x]$ and no occurrence in E, t, r

We also define the following unions of transformations:

Definition 3.10 (Extra LRPTransformation Rules Families)

1. (cpx) is the union of (cpx-in) and (cpx-e).
2. (cpcx) is the union of (cpcx-in) and (cpcx-e).
3. (ucp) is the union of (ucp1), (ucp2) and (ucp3).

(cpx) copies variables, while (cpcx) copies constructor applications, abstracting over the arguments using variables. (abs) and (abse) both abstract subexpressions by putting the arguments of constructor applications in `letrec`-environments. (ucp) is a (cp) into a unique occurrence of the variable.

Also variations of transformation rules are needed:

Definition 3.11 (Variations of LRPTransformation Rules)

(case-cx) $\texttt{letrec}\ x = (c_{K,i}\ x_1 \ldots x_n), E\ \texttt{in}$
$\mathbb{C}[\texttt{case}_K\ x\ \texttt{of}\ \{((c_{K,i}\ y_1\ \ldots\ y_n) \to s)\ \ldots\}]$
$\to \texttt{letrec}\ x = (c_{K,i}\ x_1\ \ldots\ x_n), E\ \texttt{in}$
$\mathbb{C}[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]$

(case-cx) $\texttt{letrec}\ x = (c_{K,i}\ x_1\ \ldots\ x_n), E$
$y = \mathbb{C}[\texttt{case}_K\ x\ \texttt{of}\ \{((c_{K,i}\ y_1\ \ldots\ y_n) \to s)\ \ldots\}]\ \texttt{in}\ r$
$\to \texttt{letrec}\ x = (c_{K,i}\ x_1\ \ldots\ x_n), E,$
$y = \mathbb{C}[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]\ \texttt{in}\ r$

(case-cx) in all other cases: like (case)

(case*) is defined as (case) if the scrutinized data expression is of the form $(c\ s_1\ \ldots\ s_n)$, where $(s_1, \ldots, s_n)$ is not a tuple of different variables, and otherwise it is (case-cx).

(gc=) $\texttt{letrec}\ x = y, y = s, E\ \texttt{in}\ r\ \to\ \texttt{letrec}\ y = s, E\ \texttt{in}\ r$
where $x \notin \mathit{FV}(s, E, r)$ and $y = s$ cannot be garbage collected

(caseId) $(\texttt{case}_K\ s\ \texttt{of}\ \{(\mathit{Pat}_{K,1} \to \mathit{Pat}_{K,1})\ \ldots\ (\mathit{Pat}_{K,|D_K|} \to \mathit{Pat}_{K,|D_K|})\}) \to s$

(case-cx) and (case*) are variants of (case) with an optimization w.r.t. the let-bindings, if the scrutinized data expression is of the form $(c\,x_1\ \ldots\ x_n)$. (gc=) is a special variant of (gc) where only a single variable-to-variable-binding is removed. (caseId) is a typed transformation that eliminates trivial `case`-expressions.

Also the following special transformation rules are needed:

Definition 3.12 (Special LRPTransformation Rules)

(cpS)	is (cp) restricted such that only surface contexts $\mathbb{S}$ for the target context $\mathbb{C}$ are permitted
(cpcxT)	is (cpcx) restricted such that only top contexts $\mathbb{T}$ for the target context $\mathbb{C}$ are permitted
(cse)	$\texttt{letrec}\ x = s, y = s, E\ \texttt{in}\ r \to \texttt{letrec}\ x = s, E[x/y]\ \texttt{in}\ r[x/y]$ where $x \notin \mathit{FV}(s)$
(soec)	changing the sequence of evaluation due to strictness knowledge by inserting `seq`

(cpS) and (cpcxT) are the usual transformations restricted to surface and top contexts.

(cse) means common subexpressions elimination, where common subexpressions are shared by a `letrec`-binding and only the binding-variable is used to reference the corresponding subexpression. (soec) is a correct change of the evaluation order using strictness knowledge by inserting `seq`-expressions.

In the following subsections the space behavior of several transformations is analyzed. For most of the proofs forking diagrams between transformation steps and normal-order reduction steps and induction are used (see Section 2.5), where the computation of diagrams is simplified by the context lemmas for space improvement and equivalence (see Section 3.3). Moreover we can use LRgc for easier forking diagrams and transfer the result to LRPgc:

Proposition 3.5 (Relation between LRgc and LRPgc w.r.t. Space Improvements)
Let Q be a transformation in LRgc and let Q^P be the corresponding transformation in LRPgc. We assume that Q^P does not change the type of expressions in LRPgc. We also assume that for the type-erased relation it holds that $\varepsilon(Q^P) \subseteq Q$. Then the following holds:

1. If Q is a maximal space improvement then also Q^P is a maximal space improvement.
2. If Q is a maximal space equivalence then also Q^P is a maximal space equivalence.

Proof
This is obvious, since the size measure is the same and since every type erased context from LRPgc is also an untyped context.

3.4.1 Space Improvement Property of (lbeta), (seq-c) and (case-c)

To show that (lbeta), (seq-c) and (case-c) are maximal space improvements, we first need the following lemma about the decrease of the size:

Lemma 3.8 ((lbeta), (seq) and (case) Strictly Decrease the Size)
The (case)-, (lbeta)- and (seq)-transformations in any context strictly decrease the `size`.

Proof
For (lbeta)-, (seq)- and (case-c)-reductions the claim is trivial.

For (case-e)- and (case-in)-reductions w.r.t. constructors with an arity ≥ 1, this also holds, where it is exploited that variables do not count in the measure.

Theorem 3.1 (Space Improvement Property of (lbeta), (seq-c) and (case-c))
The transformations (lbeta), (seq-c) and (case-c) are maximal space improvements.

Proof
We apply the context lemma for maximal space improvements (Lemma 3.6):

Let $s \xrightarrow{a} t$ where $a \in \{(\text{lbeta}), (\text{seq-c}), (\text{case-c})\}$ and the transformation is w.l.o.g. on the top of the expression s. The precondition $FV(t) \subseteq FV(s)$ is satisfied. Let $\mathbb{R}$ be a reduction context. We consider the cases for $\mathbb{R}[s] \xrightarrow{a} \mathbb{R}[t]$, which is normal-order for LR, but since (gc) may be also applicable, it may be not an LRPgc-reduction. An analysis of cases shows that the following diagram is valid:

$$\begin{tikzcd}
\mathbb{R}[s] \arrow[rr, "a"] \arrow[d, dashed, "LRPgc{,}gc{,}*"] & & \mathbb{R}[t] \arrow[dl, dashed, "gc{,}*"'] \arrow[dd, dashed, "LRPgc{,}gc{,}*"] \\
\mathbb{R}'[s] \arrow[r, dashed, "LRPgc{,}a"'] & \mathbb{R}'[t] \arrow[dr, dashed, "LRPgc{,}gc{,}*"'] & \\
& & \mathbb{R}''[t]
\end{tikzcd}$$

Lemma 3.8 shows that $\texttt{size}(\mathbb{R}[s]) \geq \texttt{size}(\mathbb{R}[t]) \geq \texttt{size}(\mathbb{R}''[t])$ and also all intermediate expressions in the diagram have a smaller size than $\texttt{size}(\mathbb{R}[s])$.

The definition of *spmax* implies *spmax*$(\mathbb{R}[s]) \geq$ *spmax*$(\mathbb{R}[t])$ for all reduction contexts $\mathbb{R}$. An application of Lemma 3.6 shows the claim.

3.4.2 Space Improvement Property of (gc)

In this section we prove that (gc) is a maximal space improvement.

Theorem 3.2 (Space Improvement Property of (gc))
The transformation (gc) is a maximal space improvement.

Proof
Let $s_0 \xrightarrow{gc} t_0$ in the empty context. We show that the requirements of the context lemma for maximal space improvement (see Lemma 3.6) are fulfilled:

Let $\mathbb{R}$ be a reduction context. Then we consider the reductions of $\mathbb{R}[s_0]$ and $\mathbb{R}[t_0]$. Obviously $\texttt{size}(\mathbb{R}[s_0]) \geq \texttt{size}(\mathbb{R}[t_0])$. The diagrams in Lemma 3.2 can be applied to show that *spmax*$(\mathbb{R}[s_0]) \geq$ *spmax*$(\mathbb{R}[t_0])$ by induction on the number of LRPgc-reductions.

Let $s = \mathbb{R}[s_0]$ and $t = \mathbb{R}[t_0]$. Then $s \xrightarrow{\mathbb{T},gc} t$. The conditions $\texttt{size}(s_0) \geq \texttt{size}(t_0)$ and $FV(s_0) \supseteq FV(t_0)$ are satisfied. We show that *spmax*$(s) \geq$ *spmax*(t) by induction on the following measure: (i) the number of *LCSC*-reductions, (ii) the measure μ_{lll} and (iii) the measure $\texttt{synsize}$.

We make a case analysis along the diagrams in Lemma 3.2.

- If s is an LRPgc-WHNF, then t is also an LRPgc-WHNF, since $s \xrightarrow{gc} t$ and *spmax*$(s) \geq$ *spmax*(t) holds.
- If $s \xrightarrow{LRPgc,gc} s'$ and $s \xrightarrow{gc} t$ are the same reductions (i.e. the situation of diagram 3), then clearly *spmax*$(s) \geq$ *spmax*(s') holds.
- If $s \xrightarrow{LRPgc,a} s'$ for $a \in$ *LCSC*, then for diagram 1 we can apply the induction hypothesis along the chain $s' \xrightarrow{\mathbb{T},gc,*} t'$ using Lemma 3.3. Hence we obtain *spmax*$(s') \geq$ *spmax*(t').

Since $\texttt{size}(s) \geq \texttt{size}(t)$, we obtain *spmax*$(s) \geq$ *spmax*(t) in this case. For diagram 4, we also get *spmax*$(s') \geq$ *spmax*(t') and also $\texttt{size}(s) \geq \texttt{size}(t) \geq \texttt{size}(t_1)$ and thus the claim holds.
Note that *spmax*$(s) \geq$ *spmax*(s') may be wrong for (cp)-reductions.

- If $s \xrightarrow{LRPgc,lll} s'$, then diagrams 1 and 2 are relevant. For the first diagram, we can apply the induction hypothesis to s' since the measure μ_{lll} gets strictly smaller. Since (gc)-reductions keeps the *LCSC*-number and do not increase the μ_{lll}-measure, the induction hypothesis can be applied to the reduction chain and we obtain *spmax*$(s') \geq$ *spmax*(t'). Also $\texttt{size}(s) \geq \texttt{size}(t)$ holds and the claim *spmax*$(s) \geq$ *spmax*(t) is proved. The reasoning is similar but simpler for diagram 2.
- If $s \xrightarrow{LRPgc,gc} s'$, then diagrams 1, 2 and 3 are relevant. For the first diagram, we can apply the induction hypothesis, since the *LCSC*-number is not changed in the diagram, the μ_{lll}-measure decreases along the reductions in the diagram and also the syntactical size is strictly decreased from s to s'. Hence we can apply the induction hypothesis to s' and obtain *spmax*$(s') \geq$ *spmax*(t'). Then it is easy to see that *spmax*$(s) \geq$ *spmax*(t) holds. In case of diagram 2, the reasoning is similar but simpler and diagram 3 is obvious.

An application of Lemma 3.6 now shows the claim.

3.4.3 Space Improvement Property of (lll)

To show that (lll) is a maximal space improvement, we consider the forking diagrams for (lll) in the calculus LR:

$$
\begin{array}{lcl}
\xleftarrow{LR,lbeta} \cdot \xrightarrow{lll} & \rightsquigarrow & \xrightarrow{lll} \cdot \xleftarrow{LR,lbeta} \\
\xleftarrow{LR,cp} \cdot \xrightarrow{lll} & \rightsquigarrow & \xrightarrow{lll} \cdot \xleftarrow{LR,cp} \\
\xleftarrow{LR,lll} \cdot \xrightarrow{lll} & \rightsquigarrow & \xrightarrow{lll} \cdot \xleftarrow{LR,lll} \\
\xleftarrow{LR,lll} \cdot \xrightarrow{lll} & \rightsquigarrow & \cdot \\
\xleftarrow{LR,lll} \cdot \xrightarrow{lll} & \rightsquigarrow & \xrightarrow{lll} \cdot \xrightarrow{lll} \cdot \xleftarrow{LR,lll} \\
\xleftarrow{LR,case} \cdot \xrightarrow{lll} & \rightsquigarrow & \xrightarrow{lll} \cdot \xleftarrow{LR,case} \\
\xleftarrow{LR,case} \cdot \xrightarrow{lll} & \rightsquigarrow & \xleftarrow{LR,case} \\
\xleftarrow{LR,seq} \cdot \xrightarrow{lll} & \rightsquigarrow & \xrightarrow{lll} \cdot \xleftarrow{LR,seq} \\
\xleftarrow{LR,seq} \cdot \xrightarrow{lll} & \rightsquigarrow & \xleftarrow{LR,seq} \\
WHNF \cdot \xrightarrow{lll} & \rightsquigarrow & WHNF
\end{array}
$$

The double $\xrightarrow{lll}$ occurs in the following case:

$$
\begin{array}{ll}
 & (\texttt{letrec}\ E_1\ \texttt{in}\ (\texttt{letrec}\ E_2\ \texttt{in}\ s))\ t \\
\xrightarrow{llet-in} & (\texttt{letrec}\ E_1, E_2\ \texttt{in}\ s)\ t \\
\xrightarrow{lapp} & (\texttt{letrec}\ E_1, E_2\ \texttt{in}\ (s\ t))
\end{array}
$$

(llet) never enables a (LRPgc,gc2):

Lemma 3.9 ((llet) does not enable (LRPgc,gc2))
An (llet)-transformation may enable (LRPgc,gc1) but not (LRPgc,gc2).

Proof
The only situation where this occurs is if a `letrec`-environment is shifted to the top and if the shifted `letrec`-environment contains garbage.

Now we show that (lll) is a maximal space improvement:

Theorem 3.3 (Space Improvement Property of (lll))
The transformation (lll) is a maximal space improvement.

Proof
A complete set of forking diagrams for (lll) w.r.t. space improvement can be developed using the diagrams above and summarized in three diagrams:

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{T},lll"] \arrow[ddd, "LRPgc,a"'] & & s' \arrow[d, dashed, "LRPgc,gc,0\vee 1"] \\
& a \neq gc & s'_1 \arrow[d, dashed, "LRPgc,a,0\vee 1"] \\
& & s'_2 \arrow[d, dashed, "LRPgc,gc,0\vee 1"] \\
s_3 \arrow[r, dashed, "\mathbb{T},lll,*"'] & s''_3 \arrow[r, dashed, "\mathbb{T}\vee LRPgc,gc,0\vee 1"'] & s'_3
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{T},lll"] \arrow[d, "LRPgc,gc"'] & & s' \arrow[d, dashed, "LRPgc,gc"] \\
s_1 \arrow[r, dashed, "\mathbb{T}\vee LRPgc,lll,0\vee 1"'] & s''_1 \arrow[r, dashed, "\mathbb{T}\vee LRPgc,gc,0\vee 1"'] & s'_1
\end{tikzcd}$$

$$\begin{tikzcd}
s\ (\mathit{WHNF}) \arrow[r, "\mathbb{T},lll"] & s' \arrow[d, dashed, "LRPgc,gc,0\vee 1"] \\
& s'_1\ (\mathit{WHNF})
\end{tikzcd}$$

We will apply the context lemma for top-contexts (Lemma 3.6 and Proposition 3.3). Therefore we will employ they forking diagrams above for (lll) in top contexts.

Let s be a closed expression with $s{\downarrow}_{LRPgc}$ and $s \xrightarrow{\mathbb{T},lll} s'$. We show by induction that for all top contexts $\mathbb{T}$, we have *spmax*$(\mathbb{T}[s']) \leq$ *spmax*$(\mathbb{T}[s])$. The measure for induction is (μ_1, μ_2, μ_3), ordered lexicographically, where μ_1 is the number of *LCSC*-reductions of s to a WHNF, μ_2 is μ_{lll} and μ_3 is the syntactical size.

The first two assumptions of the context lemma are satisfied, i.e. $\mathtt{size}(s) \geq \mathtt{size}(s')$ and *FV*$(s) \supseteq$ *FV*(s'), which can be easily checked.

In the following we use invariances and properties: Theorem 3.2 shows that (gc) is a space improvement, Lemma 3.3 shows that (gc) leaves μ_1 invariant and in (SSS08b), Theorem 2.14 it is proved that (lll) does not change μ_1.

We now go through the cases according to the diagrams.

- If s is an LRPgc-WHNF, then either s is an LRPgc-WHNF or one (LRPgc,gc) is sufficient to turn it into an LRPgc-WHNF. In both cases, $\mathit{spmax}(s) \geq \mathit{spmax}(s')$ or $\mathit{spmax}(s) \geq \mathit{spmax}(s'_1)$ holds.
- If $a \in \{(\text{lbeta}), (\text{case}), (\text{seq})\}$, then we can use induction and the above properties.

  The induction hypothesis is applicable to s_3, since

  $$\mu_1(s) > \mu_1(s_3) \geq \mu_1(s''_3) = \mu_1(s'_3)$$

  This shows that s'_3 improves s''_3 which in turn improves s_3.

  We have $\texttt{size}(s) \geq \texttt{size}(s_3)$ as well as

  $$\texttt{size}(s) \geq \texttt{size}(s') \geq \texttt{size}(s'_1) \geq \texttt{size}(s'_2) \geq \texttt{size}(s'_3)$$

  Hence also $\mathit{spmax}(s) \geq \mathit{spmax}(s')$.
- If $a = (\text{cp})$, then we can also use the properties. Again, the induction hypothesis can be applied to show that s'_3 improves s''_3 which in turn improves s_3.

  We have to take into account that (cp) increases the size. Looking at the diagrams and the case analyses, we see that $\texttt{size}(s) \geq \texttt{size}(s'_1)$, since the (gc) of s removes at least the same bindings as the (gc) of s'. Since the (cp)-reductions copy the same expression, we have $\texttt{size}(s_3) \geq \texttt{size}(s'_2)$.

  Thus we have shown that also in this case $\mathit{spmax}(s) \geq \mathit{spmax}(s')$ holds.
- If $a = (\text{lll})$, then the measure of s_3, s''_3 and s'_3 is strictly smaller than s, since (lll) does not change μ_1, but strictly decreases μ_2.

  By induction s''_3 space-improves s_3 and s'_3 space-improves s''_3. Since (lll) does not increase the $\texttt{size}$, we obtain $\mathit{spmax}(s) \geq \mathit{spmax}(s')$ also in this case.
- If the first LRPgc-reduction is a (gc), then the second diagram holds and we can apply induction, since (gc) does not increase μ_1 nor μ_2, but strictly decreases the syntactical size.

Finally, we can apply Lemma 3.6 and obtain the claim that s' is a maximal space improvement of s.

3.4.4 Space Improvement Property of (seq)

To show that (seq) is a space improvement we first develop a complete set of forking diagrams:

Lemma 3.10 (Forking Diagrams for (seq))
The forking diagrams for (seq) w.r.t. maximal space improvement can be summarized in the following diagrams:

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{T}{,}seq"] \arrow[ddd, "LRPgc{,}a"'] & & s' \arrow[d, dashed, "LRPgc{,}gc{,}0\vee 1"] \\
 & a \neq gc & s'_1 \arrow[d, dashed, "LRPgc{,}a{,}0\vee 1"] \\
 & & s'_2 \arrow[d, dashed, "LRPgc{,}gc{,}0\vee 1"] \\
s_3 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}seq{,}0\vee 1"'] & s''_3 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}gc{,}0\vee 1"'] & s'_3
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T}{,}seq"] \arrow[d, "LRPgc{,}gc2"'] & s' \arrow[d, dashed, "LRPgc{,}gc2"] \\
s_1 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}seq{,}0\vee 1"'] & s'_1
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{T}{,}seq"] \arrow[d, "LRPgc{,}gc1"'] & & s' \arrow[d, dashed, "LRPgc{,}gc"] \\
s_1 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}seq{,}0\vee 1"'] & s''_1 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}gc{,}0\vee 1"'] & s'_1
\end{tikzcd}$$

Proof
We check all possibilities of $s_1 \xleftarrow{LRPgc,a} s \xrightarrow{\mathbb{T},seq} s'$ for a closed expressions s, where the (seq)-reduction is not an LRPgc-reduction.

- The expression s is not a `letrec`-expression. Then closing the reduction is represented by a square diagram.

  The reason is that s does not admit an $\xrightarrow{LRPgc,gc}$-reduction.

$$\begin{tikzcd}
s \arrow[r, "i{,}seq"] \arrow[d, "LRPgc{,}a"'] & s' \arrow[d, dashed, "LRPgc{,}a"] \\
s_1 \arrow[r, dashed, "i{,}seq"'] & s'_1
\end{tikzcd}$$

  In the following we can assume that s is a `letrec`-expression.

- $s \xrightarrow{LRPgc,a} s_1$, i.e. the first reduction is not a garbage collection and a is an (lbeta)-, (seq)-, (cp)- or (case)-reduction.
  Then s = `letrec` E_1, E_2 `in` r, where E_1 is garbage after the seq-reduction. If the `seq`-reduction does not enable a $\xrightarrow{LRPgc,gc}$-reduction, then the following two diagrams are sufficient.

$$\begin{tikzcd}
s \arrow[r, "{i,seq}"] \arrow[d, "{LRPgc,a}"'] & s' \arrow[d, dashed, "{LRPgc,a}"] \\
s_1 \arrow[r, dashed, "{i,seq}"'] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "{i,seq}"] \arrow[d, "{LRPgc,a}"'] & s' \arrow[ddl, dashed, "{LRPgc,a}"] \\
s_1 \arrow[d, "{LRPgc,seq}"'] & \\
s_2 &
\end{tikzcd}$$

If seq enables a $\xrightarrow{LRPgc,gc}$-reduction, then the diagram is:

$$\begin{tikzcd}
s \arrow[rr, "{i,seq}"] \arrow[dd, "{LRPgc,a}"'] & & s' \arrow[d, dashed, "{LRPgc,gc}"] \\
& & s'_1 \arrow[d, dashed, "{LRPgc,a}"] \\
s_1 \arrow[r, dashed, "{i,seq}"'] & \cdot \arrow[r, dashed, "gc"'] & s'_2
\end{tikzcd}$$

An example is:

$$\begin{array}{rl}
 & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ ((\lambda x.x)\ x_2) \\
\xrightarrow{LRPgc,lbeta} & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x) \\
\xrightarrow{i,seq} & \texttt{letrec}\ x_1 = 0, x_2 = 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x) \\
\xrightarrow{gc} & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x) \\
\hline
\xrightarrow{i,seq} & \texttt{letrec}\ x_1 = 0, x_2 = 0\ \texttt{in}\ ((\lambda x.x)\ x_2) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ ((\lambda x.x)\ x_2) \\
\xrightarrow{LRPgc,lbeta} & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x)
\end{array}$$

- $s \xrightarrow{LRPgc,gc2} s'$. Then the diagrams for reductions are as follows:

$$\begin{tikzcd}
s \arrow[r, "{i,seq}"] \arrow[d, "{LRPgc,gc2}"'] & s' \arrow[d, dashed, "{LRPgc,gc2}"] \\
s_1 \arrow[r, dashed, "{i,seq}"'] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "{i,seq}"] \arrow[d, "{LRPgc,gc2}"'] & s' \arrow[ddl, dashed, "{LRPgc,gc,*}"] \\
s_1 \arrow[d, "{LRPgc,seq}"'] & \\
s_2 &
\end{tikzcd}$$

An example is:

$$\begin{array}{rl}
 & \texttt{letrec}\ x_1 = 0\ \texttt{in}\ \mathbb{C}[\texttt{seq}\ x_2\ 0] \\
\xrightarrow{LRPgc,gc2} & \mathbb{C}[\texttt{seq}\ x_2\ 0] \\
\xrightarrow{i,seq} & \mathbb{C}[0] \\
\hline
\xrightarrow{i,seq} & \texttt{letrec}\ x_1 = 0\ \texttt{in}\ \mathbb{C}[0] \\
\xrightarrow{LRPgc,gc2} & \mathbb{C}[0]
\end{array}$$

In the following cases s is a `letrec`-expression and the first LRPgc-reduction of s is not a (gc2).

- If a is an (lbeta)-, (seq)-, (cp)- or (case)-reduction and the first LRPgc-reduction is a (gc), (seq) enables a (gc) and the a-reduction does not enable a (gc), then the diagram is:

$$\begin{tikzcd}
s \arrow[rr, "i,seq"] \arrow[d, "LRPgc,gc"'] & & s' \arrow[d, dashed, "LRPgc,gc"] \\
s_1 \arrow[d, "LRPgc,a"'] & & s'_1 \arrow[d, dashed, "LRPgc,a"] \\
s_2 \arrow[r, dashed, "i,seq"'] & \cdot \arrow[r, dashed, "gc"'] & s'_2
\end{tikzcd}$$

An example is:

$$\begin{array}{rl}
 & \texttt{letrec}\ x_0 = 0, x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ ((\lambda x.x)\ x_2) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ ((\lambda x.x)\ x_2) \\
\xrightarrow{LRPgc,lbeta} & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x) \\
\xrightarrow{i,seq} & \texttt{letrec}\ x_1 = 0, x_2 = 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x) \\
\xrightarrow{gc} & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x) \\
\hline
\xrightarrow{i,seq} & \texttt{letrec}\ x_0 = 0, x_1 = 0, x_2 = 0\ \texttt{in}\ ((\lambda x.x)\ x_2) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ ((\lambda x.x)\ x_2) \\
\xrightarrow{LRPgc,lbeta} & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ (\texttt{letrec}\ x = x_2\ \texttt{in}\ x)
\end{array}$$

- If a is an (lbeta)-, (seq)- or (case)-reduction and the first LRPgc-reduction is a (gc), (seq) enables a (gc), the a-reduction enables a (gc) (only a (seq) or (case) is possible), then the diagram is:

$$\begin{tikzcd}
s \arrow[rr, "i{,}seq"] \arrow[d, "LRPgc{,}gc"'] & & s' \arrow[d, dashed, "LRPgc{,}gc"] \\
s_1 \arrow[dd, "LRPgc{,}a"'] & & s_1' \arrow[d, dashed, "LRPgc{,}a"] \\
& & s_2' \arrow[d, dashed, "LRPgc{,}gc"] \\
s_3 \arrow[r, dashed, "i{,}seq"'] & \cdot \arrow[r, dashed, "gc"'] & s_3'
\end{tikzcd}$$

An example is:

$$\begin{aligned}
& \texttt{letrec}\ x_0 = 0, x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0, x_3 = 0\ \texttt{in}\ (\texttt{seq}\ x_3\ x_2) \\
\xrightarrow{LRPgc,gc}\ & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0, x_3 = 0\ \texttt{in}\ (\texttt{seq}\ x_3\ x_2) \\
\xrightarrow{LRPgc,seq}\ & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0, x_3 = 0\ \texttt{in}\ x_2 \\
\xrightarrow{i,seq}\ & \texttt{letrec}\ x_1 = 0, x_2 = 0, x_3 = 0\ \texttt{in}\ x_2 \\
\xrightarrow{gc}\ & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ x_2
\end{aligned}$$

$$\begin{aligned}
\xrightarrow{i,seq}\ & \texttt{letrec}\ x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 0\ \texttt{in}\ (\texttt{seq}\ x_3\ x_2) \\
\xrightarrow{LRPgc,gc}\ & \texttt{letrec}\ x_2 = 0, x_3 = 0\ \texttt{in}\ (\texttt{seq}\ x_3\ x_2) \\
\xrightarrow{LRPgc,seq}\ & \texttt{letrec}\ x_2 = 0, x_3 = 0\ \texttt{in}\ x_2 \\
\xrightarrow{LRPgc,gc}\ & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ x_2
\end{aligned}$$

- If a is a (llet)-reduction, the first LRPgc-reduction is a (llet) and (seq) may enable a (gc). Then the diagram is:

$$\begin{tikzcd}
s \arrow[rr, "i{,}seq"] \arrow[ddd, "LRPgc{,}llet"'] & & s' \arrow[d, dashed, "LRPgc{,}gc"] \\
& & s_1' \arrow[d, dashed, "LRPgc{,}llet"] \\
& & s_2' \arrow[d, dashed, "LRPgc{,}gc"] \\
s_3 \arrow[r, dashed, "i{,}seq"'] & \cdot \arrow[r, dashed, "gc"'] & s_3'
\end{tikzcd}$$

An example is:

$$\begin{array}{rl}
 & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,llet} & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0, x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{i,seq} & \texttt{letrec}\ x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{gc} & \texttt{letrec}\ x_2 = 0, x_3 = 0\ \texttt{in}\ (x_3\ x_2) \\
\hline
\xrightarrow{i,seq} & \texttt{letrec}\ x_1 = 0, x_2 = 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,llet} & \texttt{letrec}\ x_2 = 0, x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_2 = 0, x_3 = 0\ \texttt{in}\ (x_3\ x_2)
\end{array}$$

- If a is a (llet)-reduction, the first LRPgc-reduction is a (gc) and (seq) may enable a (gc). Then the diagram is:

$$\begin{tikzcd}
s \arrow[rr, "i,seq"] \arrow[d, "LRPgc,gc"'] & & s' \arrow[d, dashed, "LRPgc,gc"] \\
s_1 \arrow[dd, "LRPgc,llet"'] & & s'_1 \arrow[d, dashed, "LRPgc,llet"] \\
 & & s'_2 \arrow[d, dashed, "LRPgc,gc"] \\
s_3 \arrow[r, dashed, "i,seq"'] & \cdot \arrow[r, dashed, "gc"'] & s'_3
\end{tikzcd}$$

An example is:

$$\begin{array}{rl}
 & \texttt{letrec}\ x_0 = 0, x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in} \\
 & \qquad\qquad \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,llet} & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0, x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{i,seq} & \texttt{letrec}\ x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{gc} & \texttt{letrec}\ x_2 = 0, x_3 = 0\ \texttt{in}\ (x_3\ x_2) \\
\hline
\xrightarrow{i,seq} & \texttt{letrec}\ x_0 = 0, x_1 = 0, x_2 = 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_2 = 0\ \texttt{in}\ \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,llet} & \texttt{letrec}\ x_2 = 0, x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_2 = 0, x_3 = 0\ \texttt{in}\ (x_3\ x_2)
\end{array}$$

- If the binding $x_4 = 0$ is missing in the last example, then $s'_2 \xrightarrow{LRPgc,gc} s'_3$ can be omitted. Thus the summary of the last two diagrams is:

$$\begin{tikzcd}
s \arrow[rr, "i{,}seq"] \arrow[d, "(LRPgc{,}gc)^{0\vee 1}"'] & & s' \arrow[d, dashed, "LRPgc{,}gc"] \\
s_1 \arrow[dd, "LRPgc{,}llet"'] & & s'_1 \arrow[d, dashed, "LRPgc{,}llet"] \\
& & s'_2 \arrow[d, dashed, "(LRPgc{,}gc)^{0\vee 1}"] \\
s_3 \arrow[r, dashed, "i{,}seq"'] & \cdot \arrow[r, dashed, "gc"'] & s'_3
\end{tikzcd}$$

- If a is a (llet)-reduction, the first LRPgc-reduction step is (gc) and (seq) may enable a (gc), then this also leads to the following diagrams:

$$\begin{tikzcd}
s \arrow[rr, "i{,}seq"] \arrow[d, "LRPgc{,}gc"'] & & s' \arrow[dd, dashed, "LRPgc{,}gc"'] \\
s_1 \arrow[d, "LRPgc{,}llet"'] & & \\
s_2 \arrow[r, dashed, "i{,}seq"'] & \cdot \arrow[r, dashed, "gc"'] & s'_2
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[rr, "i{,}seq"] \arrow[d, "LRPgc{,}gc"'] & & s' \arrow[d, dashed, "LRPgc{,}gc"] \\
s_1 \arrow[d, "LRPgc{,}llet"'] & & s'_1 \arrow[d, dashed, "LRPgc{,}gc{,}0\vee 1"] \\
s_2 \arrow[r, dashed, "i{,}seq"'] & \cdot \arrow[r, dashed, "gc"'] & s'_2
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rr, "i{,}seq"] \arrow[dd, "LRPgc{,}llet"'] & & s' \arrow[d, dashed, "LRPgc{,}gc"] \\
& & s'_1 \arrow[d, dashed, "LRPgc{,}gc{,}0\vee 1"] \\
s_2 \arrow[r, dashed, "i{,}seq"'] & \cdot \arrow[r, dashed, "gc"'] & s'_2
\end{tikzcd}$$

Examples for the diagrams:

$$\begin{array}{rl}
 & \texttt{letrec}\ x = 0\ \texttt{in}\ (\texttt{letrec}\ y = \texttt{seq}\ x\ 0\ \texttt{in} \\
 & \qquad\qquad (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0))) \\
\xrightarrow{LRPgc,llet-in} & \texttt{letrec}\ x = 0, y = \texttt{seq}\ x\ 0\ \texttt{in}\ (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0)) \\
\xrightarrow{i,seq} & \texttt{letrec}\ x = 0, y = 0\ \texttt{in}\ (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0)) \\
\xrightarrow{gc2} & \texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0) \\
\hline
\xrightarrow{i,seq} & \texttt{letrec}\ x = 0\ \texttt{in}\ (\texttt{letrec}\ y = 0\ \texttt{in} \\
 & \qquad\qquad (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0))) \\
\xrightarrow{LRPgc,gc2} & \texttt{letrec}\ y = 0\ \texttt{in}\ (\texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0)) \\
\xrightarrow{LRPgc,gc2} & \texttt{letrec}\ v = 0\ \texttt{in}\ ((\lambda z.z)\ 0)
\end{array}$$

$$
\begin{array}{rl}
 & \texttt{letrec}\ x_0 = 0, x_1 = 0\ \texttt{in letrec}\ x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in} \\
 & \qquad\qquad\qquad \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_1 = 0\ \texttt{in letrec}\ x_2 = \texttt{seq}\ x_1\ 0\ \texttt{in} \\
 & \qquad\qquad\qquad \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,llet} & \texttt{letrec}\ x_1 = 0, x_2 = \texttt{seq}\ x_1\ 0, \texttt{in letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{i,seq} & \texttt{letrec}\ x_1 = 0, x_2 = 0\ \texttt{in letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{gc} & \texttt{letrec}\ x_2 = 0\ \texttt{in letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\hline
\xrightarrow{i,seq} & \texttt{letrec}\ x_0 = 0,\ x_1 = 0\ \texttt{in letrec}\ x_2 = 0\ \texttt{in} \\
 & \qquad\qquad\qquad \texttt{letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2) \\
\xrightarrow{LRPgc,gc2} & \texttt{letrec}\ x_2 = 0\ \texttt{in letrec}\ x_3 = 0, x_4 = 0\ \texttt{in}\ (x_3\ x_2)
\end{array}
$$

Now we show that (seq) is a maximal space improvement:

Theorem 3.4 (Space Improvement Property of (seq))
The transformation (seq) is a maximal space improvement.

Proof
This is already proved in the case of (seq-c) in Theorem 3.1.

For the general case of a seq-reduction we will apply the context lemma for top-contexts (see Lemma 3.6 and Proposition 3.3). Therefore we will employ forking diagrams for (seq) in top contexts from Lemma 3.10.

Let s be a closed expression with $s{\downarrow}_{LRPgc}$ and $s \xrightarrow{\mathbb{T},seq} s'$. We show by induction on the following measure that $s' \leq_{spmax} s$ by showing that for all top contexts $\mathbb{T}$, we have $spmax(\mathbb{T}[s']) \leq spmax(\mathbb{T}[s])$. The measure for induction is (μ_1, μ_2, μ_3), ordered lexicographically, where μ_1 is the number of *LCSC*-reductions of s to a WHNF, μ_2 is μ_{lll} and μ_3 is the syntactical size.

The first two assumptions of the context lemma are satisfied, i.e. $\texttt{size}(s) \geq \texttt{size}(s')$ and $FV(s) \supseteq FV(s')$, which can be easily checked.

In the following we use the following invariances and properties:
(i) Theorem 3.2 shows that (gc) is a space improvement, (ii) Lemma 3.3 shows that (gc) leaves μ_1 invariant and (iii) μ_1 is not increased by (seq) as proved in (SSS08b).

We now go through the cases for the diagrams:

- If s is an LRPgc-WHNF, then s' is also an LRPgc-WHNF and hence we have $\texttt{size}(s) = spmax(s)$, $\texttt{size}(s') = spmax(s')$ and $spmax(s) \geq spmax(s')$.
- $s \xrightarrow{\mathbb{T},seq} s'$ is an LRPgc-reduction step, then we have $spmax(s) = spmax(s')$ by definition.
- If $a \in \{(\text{lbeta}), (\text{case}), (\text{seq})\}$ and the LRPgc-reduction is different from the transformation, then we can use induction and the above properties.
  We are in the situation of diagram 1. The induction hypothesis is applicable to

s_3, since $\mu_1(s) > \mu_1(s_3) \geq \mu_1(s_3'') = \mu_1(s_3')$. This shows that s_3' improves s_3'' which in turn improves s_3. We have $\texttt{size}(s) \geq \texttt{size}(s_3)$ as well as $\texttt{size}(s) \geq \texttt{size}(s') \geq \texttt{size}(s_1') \geq \texttt{size}(s_2') \geq \texttt{size}(s_3')$. Hence also *spmax*$(s) \geq$ *spmax*(s').

- If a = (cp), then we can also use the properties. Again, the induction hypothesis can be applied to show that s_3' improves s_3'' which in turn improves s_3.
  We have to take into account that (cp) increases the size. Looking at the diagrams and the case analyses, we see that $\texttt{size}(s) \geq \texttt{size}(s_1')$, since the (gc) of s removes at least the same bindings as the (gc) from s'. Since the cp-reductions copy the same expression, we have $\texttt{size}(s_3) \geq \texttt{size}(s_2')$. Thus we have shown that also in this case *spmax*$(s) \geq$ *spmax*(s') holds.
- If a = (lll), then by induction s_3'' space-improves s_3, and s_3' space-improves s_3''. Since (lll) does not increase the $\texttt{size}$ (see Theorem 3.3), *spmax*$(s) \geq$ *spmax*(s') holds.
- If the first LRPgc-reduction step is a (gc2), then the second diagram holds, and we can apply induction, since (gc) does neither increase μ_1 (see Lemma 3.3) nor μ_2, but strictly decreases the syntactical size.
- If the first LRPgc-reduction step is a (gc1), then the third diagram holds and we can apply induction, since (gc) does neither increase μ_1 nor μ_2, but strictly decreases the syntactical size. Then the $\texttt{size}$ decreases along the reductions and we can reason as before.

Finally, we can apply Lemma 3.6 and obtain the claim that s' is a maximal space improvement of s.

3.4.5 Space Equivalence Property of (gc=)

The special variant of garbage collection (gc=) is a maximal space equivalence:

Theorem 3.5 (Space Equivalence Property of (gc=))
The transformation (gc=) is a maximal space equivalence.

Proof
An analysis of forking overlaps between LRPgc-reductions and gc=-transformations, shows that they (almost) commute:

$s_1 \xleftarrow{LRPgc,a} s \xrightarrow{\mathbb{T},gc=} s'$ can be joined by $s_1 \xrightarrow{\mathbb{T},gc=,0\vee 1} s_1' \xleftarrow{LRPgc,a} s'$.

We will apply the context lemma for space equivalence (Lemma 3.7), which also holds for top contexts (see Proposition 3.3).

Let $s_0 \xrightarrow{gc=} t_0$, and let $s = \mathbb{R}[s_0]$ and $s' = \mathbb{R}[t_0]$. Then $\texttt{size}(s) = \texttt{size}(s')$ as well as *FV*(s) = *FV*(s'). The equality *spmax*(s) = *spmax*(s') can easily be shown by induction on the number of LRPgc-reductions. Then an application of Lemma 3.7 shows the claim.

3.4.6 Space Equivalence Property of (cpx)

(cpx) is often used in other proofs, since the transformation is able to model subparts of other transformations and is a maximal space equivalence:

Theorem 3.6 (Space Equivalence Property of (cpx))
The transformation (cpx) is a maximal space equivalence.

Proof
Due to the context lemma it is sufficient to check forking diagrams in top contexts, however, we permit that (cpx) may copy into arbitrary contexts.

An analysis of forking overlaps between LRPgc-reductions and (cpx) transformations in top contexts shows that the following set of diagrams is complete, where all concrete (cpx)-transformations in a diagram copy from the same binding $x = y$:

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T},cpx"] \arrow[d, "LRPgc,a"'] & s' \arrow[d, dashed, "LRPgc,a"] \\
s_1 \arrow[r, dashed, "\mathbb{T},cpx,*"'] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[rr, "\mathbb{T},cpx"] \arrow[dd, "LRPgc,a\neq gc"'] & & s' \arrow[d, dashed, "LRPgc,gc,0\vee 1"] \\
& & s'_1 \arrow[d, dashed, "LRPgc,a"] \\
s_2 \arrow[r, dashed, "\mathbb{T},cpx,*"'] & \cdot \arrow[r, dashed, "\mathbb{T},gc=,0\vee 1"'] & s'_2
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{T},cpx"] \arrow[dd, "LRPgc,gc"'] & & s' \arrow[d, dashed, "LRPgc,gc,0\vee 1"] \\
& & s'_1 \arrow[d, dashed, "LRPgc,gc"] \\
s_2 \arrow[r, dashed, "\mathbb{T},cpx,*"'] & \cdot \arrow[r, dashed, "\mathbb{T},gc=,0\vee 1"'] & s'_2
\end{tikzcd}$$

We also need the diagram-property that $s_1 \xleftarrow{LRPgc,a} s \xrightarrow{\mathbb{T},gc=} s'$ can be joined by $s_1 \xrightarrow{\mathbb{T},gc=,0\vee 1} s'_1 \xleftarrow{LRPgc,a} s'$.

We will apply the context lemma for space equivalence (Lemma 3.7), which also holds for top contexts (see Proposition 3.3).

Let $s_0 \xrightarrow{cpx} t_0$, and let $s = \mathbb{T}[s_0]$ and $s' = \mathbb{T}[t_0]$. Then $\mathtt{size}(s) = \mathtt{size}(s')$ as well as $FV(s) = FV(s')$. We have to show $spmax(s) = spmax(s')$, which can be shown by an induction on the number of LRPgc-reductions of $\mathbb{T}[s_0]$, where we show that the number of LRPgc-reductions of $\mathbb{T}[s_0]$ is not greater than for $\mathbb{T}[t_0]$. Since (cpx) as well as (gc=) do not change the size, we have the same maximal space usage for s and s'. An application of Lemma 3.7 finishes the proof.

3.4.7 Space Equivalence Properties of (xch), (abs) and (abse)

In this section we show that (xch), (abs) and (abse) are maximal space equivalences.

Theorem 3.7 (Space Equivalence Properties of (xch), (abs) and (abse))
The transformations (xch), (abs) and (abse) are maximal space equivalences.

Proof
We first show that (xch) and (abs) are maximal space equivalences.

The transformations (xch) and (abs) do not change the LRPgc-WHNF property.
A complete set of forking diagrams for (xch) and (abs) in top contexts is:

$$\begin{tikzcd}
\cdot \arrow[r, "\mathbb{T}{,}xch"] \arrow[d, "LRPgc{,}a"'] & \cdot \arrow[d, dashed, "LRPgc{,}a"] \\
\cdot \arrow[r, dashed, "\mathbb{T}{,}xch"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{T}{,}xch"] \arrow[d, "LRPgc{,}a"'] & \cdot \arrow[dl, dashed, "LRPgc{,}a"] \\
\cdot &
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T}{,}abs"] \arrow[d, "LRPgc{,}a"'] & s' \arrow[d, dashed, "LRPgc{,}a"] \\
s_1 \arrow[r, dashed, "\mathbb{T}{,}abs"'] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{T}{,}abs"] \arrow[d, "LRPgc{,}a"'] & s' \arrow[dl, dashed, "LRPgc{,}a"] \\
s_1 &
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{T}{,}abs"] \arrow[d, "LRPgc{,}case"'] & s' \arrow[d, dashed, "LRPgc{,}case"] \\
s_1 \arrow[r, dashed, "\mathbb{T}{,}cpx\vee gc={,}*"'] & s'_1
\end{tikzcd}$$

These diagrams can be derived from the more general diagrams in (SSS08b). These transformations keep the number of *LCSC*-reductions.

The same proof technique as in Theorem 3.6 is used, i.e. the context lemma for space equivalence (Lemma 3.7) and induction proofs with the same measure.

First the space equivalence property of (xch) is proved using the same methods as described above and the context lemma for maximal space equivalence. The next part is the space equivalence property of (abs), which follows from the space equivalence property of (cpx) by Theorem 3.6 and (gc=) by Theorem 3.5.

(abse) can be showed analogous using the space equivalence properties of (abs), (cpx) and (gc=).

3.4.8 Space Improvement Properties of (case*) and (case)

In this section show that (case) is a maximal space improvement.

To show, that (case) is a maximal space improvement, we first show that the special variant (case*) is a maximal space improvement. The difference between (case) and (case*) are only a few applications of (cpx) and (gc=), that are already showed to be maximal space equivalences.
Moreover the forkings of (case*) are comparable to those of (seq), hence we can adapt the proof for (seq) to (case*). Now we show that (case*) is a maximal space improvement.

It follows from (SSS08b) that the (case*)-transformation is a correct program transformation, since the difference to a (case)-transformation consists of applications of (gc) and (cpx). I.e. it is easy to see that, if $s \xrightarrow{case} s_1$ and $s \xrightarrow{case*} s_2$, then $s_1 \xrightarrow{gc\vee cpx,*} s_2$.

We now develop a complete set of forking diagrams for (case*):

Lemma 3.11 (Forking Diagrams for (case*))
The forking diagrams for (case*) w.r.t. maximal space improvement are summarized in the following diagrams:

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{T}{,}case*"] \arrow[ddd, "LRPgc{,}a"'] & & s' \arrow[d, dashed, "LRPgc{,}gc{,}0\vee1"] \\
& a \neq gc & s'_1 \arrow[d, dashed, "LRPgc{,}a{,}0\vee1"] \\
& & s'_2 \arrow[d, dashed, "LRPgc{,}gc{,}0\vee1"] \\
s_3 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}case*{,}0\vee1"'] & s''_3 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}gc{,}0\vee1"'] & s'_3
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T}{,}case*"] \arrow[d, "LRPgc{,}gc2"'] & s' \arrow[d, dashed, "LRPgc{,}gc2"] \\
s_1 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}case*{,}0\vee1"'] & s'_1
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{T}{,}case*"] \arrow[d, "LRPgc{,}gc1"'] & & s' \arrow[d, dashed, "LRPgc{,}gc"] \\
s_1 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}case*{,}0\vee1"'] & s''_1 \arrow[r, dashed, "\mathbb{T}\vee LRPgc{,}gc{,}0\vee1"'] & s'_1
\end{tikzcd}$$

Proof
By inspecting all cases. The diagrams for (case*) are very similar to the ones for (seq) (see Lemma 3.10), since the effects of overlaps are comparable.

Using these diagrams, we show the space improvement property of (case*):

Theorem 3.8 (Space Improvement Property of (case*))
The transformation (case*) is a maximal space improvement.

Proof
The forking diagrams for (case*) in Lemma 3.11 are similar to the ones in Lemma 3.10. Thus the proof is almost the same as the proof of Theorem 3.4, where (seq) has to replaced by (case*) and we use that (case*) also decreases the `size` of expressions.

Now we can show that (case) is a maximal space improvement:

Theorem 3.9 (Space Improvement Property of (case))
The transformation (case) is a maximal space improvement.

Proof
Either (case*) is the same as (case) or (case*) is the same as (case) followed by several (cpx)- and (gc=)-transformations. Since (case*) is a maximal space improvement by Theorem 3.8 and moreover (cpx) and (gc=) are maximal space equivalences (see Theorem 3.6 and Theorem 3.5), the claim holds.

3.4.9 Space Equivalence Property of (ucp)

The transformation (ucp) is important for the translation to machine language (see Section 2.4.1). First we develop a set forking diagrams:

Lemma 3.12 (Forking Diagrams for (ucp))

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T},ucp"] \arrow[d, "LRPgc,a"'] & s' \arrow[d, dashed, "LRPgc,a"] \\
s_1 \arrow[r, dashed, "\mathbb{T},ucp"] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{T},ucp"] \arrow[d, "LRPgc,a"'] & s' \arrow[dl, dashed, "LRPgc,a"] \\
s_1 &
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[d, "LRPgc,cp"'] \arrow[dd, bend left, "\mathbb{T},ucp"] \\
s_2 \arrow[d, "LRPgc,gc"'] \\
s_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{T},ucp"] \arrow[d, "LRPgc,a"'] & s' \arrow[ddl, dashed, "LRPgc,a"] \\
s_2 \arrow[d, "LRPgc,gc"'] & \\
s_1 &
\end{tikzcd}$$

$$a \in \{seq, case\}$$

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{T},ucp"] \arrow[d, "LRPgc,case"'] & & s' \arrow[dd, dashed, "LRPgc,case"] \\
s_1 \arrow[d, "LRPgc,gc="'] & & \\
s_2 \arrow[r, dashed, "\mathbb{T},cpx,*"] & s''_2 \arrow[r, dashed, "\mathbb{T},gc=,*"] & s'_2
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{T},ucp"] \arrow[d, "LRPgc,lll"'] & s' \arrow[dd, dashed, "LRPgc,lll\vee gc,*"] \\
s_2 \arrow[d, dashed, "LRPgc,lll\vee gc,*"'] & \\
s_1 \arrow[r, dashed, "\mathbb{T},ucp,0\vee 1"] & s'_1
\end{tikzcd}$$

where the left (LRPgc,case) triggers a (gc)

Proof
The diagrams are adapted from (SSS08b) as follows:

(i) in diagram 3 the case `letrec` $x = \lambda y.r, E$ `in` $\mathbb{R}[x] \to$ `letrec` E `in` $\mathbb{R}[\lambda y.r]$ with an application of (cp) followed by a (gc) is covered and (ii) in diagram 6 also an intermediate (gc) may be triggered.

We now go through the cases for diagrams 4, 5 and 6 explicitly:

An example for diagram 4:

$$\begin{array}{rl}
 & \texttt{letrec}\ x = a, E\ \texttt{in seq}\ (c\ x)\ b \\
\xrightarrow{LRPgc,seq} & \texttt{letrec}\ x = a, E\ \texttt{in}\ b \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ E\ \texttt{in}\ b \\
\hline
\xrightarrow{\mathbb{T},ucp} & \texttt{letrec}\ E\ \texttt{in seq}\ (c\ a)\ b \\
\xrightarrow{LRPgc,seq} & \texttt{letrec}\ E\ \texttt{in}\ b
\end{array}$$

For diagram 5 we have the following case:

$$\begin{array}{rl}
 & \texttt{letrec}\ x = c\,r_i, E\ \texttt{in case}\ x\ \texttt{of}\ \{(c\,y_i \to a)\ \ldots\} \\
\xrightarrow{LRPgc,case} & \texttt{letrec}\ x = c\,r_i, z_i = r_i, E\ \texttt{in}\ (\texttt{letrec}\ y_i = z_i\ \texttt{in}\ a) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ z_i = r_i, E\ \texttt{in}\ (\texttt{letrec}\ y_i = z_i\ \texttt{in}\ a) \\
\xrightarrow{cpx \vee gc,*} & \texttt{letrec}\ E\ \texttt{in}\ (\texttt{letrec}\ y_i = r_i\ \texttt{in}\ a) \\
\hline
\xrightarrow{\mathbb{T},ucp} & \texttt{letrec}\ E\ \texttt{in case}\ (c\,r_i)\ \texttt{of}\ \{(c\,y_i \to a)\ \ldots\} \\
\xrightarrow{LRPgc,case} & \texttt{letrec}\ E\ \texttt{in}\ (\texttt{letrec}\ y_i = r_i\ \texttt{in}\ a)
\end{array}$$

For diagram 6 a prototypical case is:

$$\begin{array}{rl}
 & \texttt{letrec}\ x = (\texttt{letrec}\ E_1\ \texttt{in}\ c), E_2\ \texttt{in}\ (\texttt{letrec}\ E_3\ \texttt{in}\ x) \\
\xrightarrow{LRPgc,llet} & \texttt{letrec}\ x = c, E_1, E_2\ \texttt{in}\ (\texttt{letrec}\ E_3\ \texttt{in}\ x) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x = c, E_1, E_2'\ \texttt{in}\ (\texttt{letrec}\ E_3\ \texttt{in}\ x) \\
 & \vdots \\
 & \texttt{letrec}\ E_2', E_3', E_1'\ \texttt{in}\ c \\
\hline
\xrightarrow{\mathbb{T},ucp} & \texttt{letrec}\ E_2\ \texttt{in}\ (\texttt{letrec}\ E_3\ \texttt{in}(\texttt{letrec}\ E_1\ \texttt{in}\ c)) \\
\xrightarrow{LRPgc,llet} & \texttt{letrec}\ E_2, E_3\ \texttt{in}\ (\texttt{letrec}\ E_1\ \texttt{in}\ c) \\
 & \vdots \\
 & \texttt{letrec}\ E_2', E_3', E_1'\ \texttt{in}\ c
\end{array}$$

Now we can show that (ucp) is a maximal space equivalence:

Theorem 3.10 (Space Equivalence Property of (ucp))
The transformation (ucp) is a maximal space equivalence.

Proof
We apply the forking diagrams of Lemma 3.12. The proof technique is to apply the context lemma for maximal space equivalences (Lemma 3.7).

Let s_0, s_0' be expressions with $s = \mathbb{T}[s_0] \xrightarrow{ucp} \mathbb{T}[s_0'] = s'$. It is easy to see that $\texttt{size}(s) = \texttt{size}(s')$ and $FV(s) = FV(s')$.

We show the prerequisite $spmax(s) = spmax(s')$ for the context lemma by induction on the following measure of s: (i) μ_1, the number of *LCSC*-reductions, (ii) the measure $\mu_{lll}(s)$ and (iii) the measure $\texttt{synsize}$.

If s is a LRPgc-WHNF, then the claim holds, since for $s \xrightarrow{ucp} t$: s is a WHNF if and only if t is a WHNF and since (ucp) does not change the size.

If diagram 1 is applicable, then the induction hypothesis can be applied to s_1 and thus $spmax(s_1) = spmax(s_1')$. Since $\texttt{size}(s) = \texttt{size}(s')$, also $spmax(s) = spmax(s')$ holds.

If diagram 2 is applicable, it is obvious, since $\texttt{size}(s) = \texttt{size}(s')$.

If diagram 3 is applicable, then the induction hypothesis can be applied to s_1 and thus *spmax*(s) = *spmax*(s_1), due to the definition of *spmax* (compare Definition 3.6).

If diagram 4 is applicable, then *spmax*(s) = *spmax*(s'), since $\texttt{size}(s) = \texttt{size}(s')$ and (seq), (case) and (gc) do not increase the size.

If diagram 5 applies, then $\texttt{size}(s) = \texttt{size}(s')$ and *spmax*(s_2) = *spmax*(s_2'), since (cpx) and (gc=) are space equivalences (see Theorem 3.6 and Theorem 3.5), hence *spmax*(s) = *spmax*(s').

For diagram 6, the claim is obvious if $s_1 = s_1'$. If $s_1 \xrightarrow{\mathbb{T},ucp} s_1'$, then the induction hypothesis can be applied to s_1 and thus *spmax*(s_1) = *spmax*(s_1'). Since we have $\texttt{size}(s) = \texttt{size}(s')$ and $\texttt{size}(s_1)$, $\texttt{size}(s_1')$ and $\texttt{size}(s_2)$ are not greater than $\texttt{size}(s)$, the claim is proved.

Now we can apply Lemma 3.7, which shows the claim.

3.4.10 Space-Property of (cpcx)

(cpcx) is neither a maximal space improvement nor a maximal space equivalence. First we give a set forking diagrams:

Lemma 3.13 (Forking Diagrams for (cpcx))
The transformation (cpcx) does not change the LRPgc-WHNF property.
A complete set of forking diagrams for (cpcx) in top contexts is:

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T},cpcxT"] \arrow[d, "LRPgc{,}a"'] & s' \arrow[d, dashed, "LRPgc{,}a"] \\
s_1 \arrow[r, dashed, "\mathbb{T},cpcxT{,}0\vee 1"] & s_1'
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{T},cpcxT"] \arrow[dr, dashed, "abs"] \arrow[dddd, "LRPgc{,}a"'] & & s' \arrow[dd, dashed, "LRPgc{,}gc"] \\
& \cdot \arrow[dr, dashed, "\mathbb{T},ucp"] & \\
& & s_2' \arrow[dd, dashed, "LRPgc{,}a"] \\
& \cdot \arrow[dr, dashed, "\mathbb{T},ucp"] & \\
s_1 \arrow[ur, dashed, "abs"] \arrow[r, dashed, "\mathbb{T},cpcxT"] & s_1'' \arrow[r, dashed, "gc"] & s_1'
\end{tikzcd}$$

where $a \neq gc$

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T},cpcxT"] \arrow[d, "LRPgc{,}a"'] & s' \arrow[d, dashed, "LRPgc{,}a"] \\
s_1 \arrow[r, dashed, "\mathbb{T},abs"] & s_1'
\end{tikzcd}$$

$a \in \{seq, case\}$

$$\begin{tikzcd}
s \arrow[rrr, "\mathbb{T},cpcxT"] \arrow[d, "LRPgc{,}case"'] & & & s' \arrow[d, dashed, "LRPgc{,}case"] \\
s_1 \arrow[r, dashed, "\mathbb{T},cpcxT"'] & s_2 \arrow[r, dashed, leftrightarrow, "\mathbb{T},cpx{,}*"'] & \cdot \arrow[r, dashed, leftrightarrow, "\mathbb{T},gc{=}{,}*"'] & s_1'
\end{tikzcd}$$

Proof
The diagrams can be derived from the cases and examples in (SSS08b) and omitting the diagram, where the copy target is within an abstraction.

A (cpcxT) may trigger a new garbage collection, which have to be covered by the diagrams. The diagrams are explicit and have more arrows for more information on the the cases, since they are used to prove space properties.

Diagram 1 is the non-interfering case, also with (LRPgc,gc) and the case where the $x = (c\ \overrightarrow{t})$-binding is removed.

Diagram 2 covers the case where the $x = (c\ \overrightarrow{t})$-binding will become garbage after the copy.

Diagram 3 covers the case where the target position of (cpcxT) is removed by the (LRPgc,a)-reduction.

Diagram 4 covers the case where (cpcxT) copies into the variable-chain of the normal-order case-reduction or somewhere else.

It is easy to see that (cpcxT) is not a maximal space improvement, but we want to analyze (cpcxT) applied in top contexts:

Proposition 3.6 (Space-Property of (cpcxT) in Top Contexts)
If $s \xrightarrow{\mathbb{T},cpcxT} s'$, then *spmax*$(s) \leq$ *spmax*$(s') \leq$ *spmax*$(s) + 1$.

Proof
The inequality is shown by an induction argument using the commuting variants of the diagrams in Lemma 3.13.

We show the claim by induction on the following measure of s: (i) the number of *LCSC*-reductions, (ii) the measure μ_{lll} and (iii) the measure `synsize`.

If s, s' are LRPgc-WHNFs, then the claim holds, since $\texttt{size}(s) + 1 = \texttt{size}(s')$. If s is an LRPgc-WHNF and s' is not an LRPgc-WHNF, then only one binding of size 1 may be garbage collected, hence *spmax*$(s) + 1 =$ *spmax*(s'). Otherwise s is LRPgc-reducible and we have to check all diagrams:

Assume the first diagram is applicable. If $s_1 = s_1'$, then there are two cases: If *spmax*$(s_1) \geq \texttt{size}(s) + 1$, then *spmax*$(s_1) =$ *spmax*$(s) =$ *spmax*(s'). Otherwise, if *spmax*$(s_1) \leq \texttt{size}(s)$, then *spmax*$(s) = \texttt{size}(s)$ and *spmax*$(s') = \texttt{size}(s) + 1$.

If $s_1 \neq s_1'$, then the induction hypothesis is applicable. The computation of the maximum yields *spmax*$(s) \leq$ *spmax*$(s') \leq$ *spmax*$(s) + 1$.

In the case of diagram 2, Theorem 3.10 and Theorem 3.7 show *spmax*$(s) =$ *spmax*(s_2') and *spmax*$(s_1) =$ *spmax*(s_1'). We have *spmax*$(s) \leq$ *spmax*$(s') \leq$ *spmax*$(s) + 1$, since $\texttt{size}(s') = \texttt{size}(s) + 1$.

In the case of diagram 3 we obtain *spmax*$(s_1) =$ *spmax*(s_1') by Theorem 3.7 and since $\texttt{size}(s') = \texttt{size}(s) + 1$, this shows the claim.

In the fourth diagram, the induction hypothesis can be applied. This together with Theorem 3.6 and Theorem 3.5 shows *spmax*$(s_1) \leq$ *spmax*$(s_2) \leq$ *spmax*$(s_1) + 1$ and *spmax*$(s_2) =$ *spmax*(s_1'). Computing the maximum shows the claim.

3.4.11 Space Improvement Property of (caseId)

In this section we analyze (caseId) (see Definition 3.11). This transformation can be seen as typed maximal space improvement. If (caseId) is used in LRgc, then it might not be correct, since LRgc has no typing. However all proofs are still done using LRgc, since we only need to require that specific (caseId)-transformations are correct and still use the same proof method as in the sections before.

The transformation (caseId) is the heart of other transformations that are only correct under typing. Examples for such transformations are (`map` $\lambda x.x$) → `id`, `foldr` (:) [] → `id` and `filter` ($\lambda x.$`True`) → `id` (see Section 2.6 for needed definitions of functions).

(caseId) is correct in LRP as proved in (SSS16b), but not in LR, which can be seen by trying the case $s = \lambda x.t$. We only consider transformation instances $s_1 \xrightarrow{caseId} s_2$ in LRgc, where s_1 and s_2 are contextual equivalent. In this case the instance of the (caseId)-transformation is called *correct* in LRgc.

First we give a complete set of forking diagrams for (caseId):

Lemma 3.14 (Forking Diagrams for (caseId))
A complete set of forking diagrams for the correct instances of (caseId) is as follows. Whenever the starting (caseId)-transformation is correct, then the other (caseId)-transformations in the diagrams are also correct.

$$\begin{tikzcd}
s \arrow[r, "{\mathbb{T},caseId}"] \arrow[d, "{LRPgc,lcase}"'] & s' \\
s_1 \arrow[ur, dashed, "{\mathbb{T},caseId}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rrr, "{\mathbb{T},caseId}"] \arrow[ddd, "{LRPgc,case}"'] & & & s' \arrow[dl, dashed, "{\mathbb{T},cpcxT,0\vee 1}"] \\
& & s_3 & \\
& s_2 \arrow[ur, dashed, "{\mathbb{T},gc{=},*}"'] & & \\
s_1 \arrow[ur, dashed, "{\mathbb{T},cpx,*}"'] & & &
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[d, "{LRPgc,case}"'] \arrow[d, dashed, bend left, "{\mathbb{T},caseId}"] \\
s_1
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rr, "{\mathbb{T},caseId}"] \arrow[dd, "{LRPgc,a}"'] & & s' \arrow[d, dashed, "{LRPgc,gc,0\vee 1}"] \\
& & s_2' \arrow[d, dashed, "{LRPgc,a}"] \\
s_1 \arrow[r, dashed, "{\mathbb{T},caseId,0\vee 1}"'] & s_3 \arrow[r, dashed, "{gc,*}"'] & s_1'
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rr, "{\mathbb{T},caseId}"] \arrow[d, "{LRPgc,gc}"'] & & s' \arrow[d, dashed, "{LRPgc,gc}"] \\
s_1 \arrow[r, dashed, "{\mathbb{T},caseId,0\vee 1}"'] & s_1' \arrow[r, dashed, "{gc,*}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "{\mathbb{T},caseId}"] \arrow[d, "{LRPgc,case{-}c}"'] & s' \\
s_1 \arrow[ur, dashed, "{\mathbb{T},ucp}"'] &
\end{tikzcd}$$

Proof
These are adaptations from (SSS16b).

Now we can show, that for correct instances, (caseId) is a maximal space improvement.

Theorem 3.11 (Space Improvement Property of Correct Instances of (caseId))
Correct instances of the (caseId)-transformation are maximal space improvements.

Proof
We apply the context lemma for maximal space improvements (Lemma 3.6) and the diagrams in Lemma 3.14.

Let s, s' be expressions with $s \xrightarrow{\mathbb{T},caseId} s'$. We show that the prerequisites for the context lemma for maximal space improvements are fulfilled:

The conditions $\mathit{FV}(s) \supseteq \mathit{FV}(s')$ and $\mathtt{size}(s) \geq \mathtt{size}(s')$ obviously hold. We show the third condition $s' \leq_{R,\mathit{spmax}} s$ by induction on the following measure of s: (i) the number of *LCSC*-reductions μ_1, (ii) the measure μ_{lll} and (iii) the measure $\mathtt{synsize}$.

If s is a LRPgc-WHNF, then s' is also a LRPgc-WHNF and the claim holds, since $\mathtt{size}(s) \geq \mathtt{size}(s')$. Now s has a LRPgc-reduction and we check each applicable diagram:

If diagram 1 is applicable, then the induction hypothesis can be applied to s_1 and we obtain $\mathit{spmax}(s_1) \geq \mathit{spmax}(s')$. Since $\mathtt{size}(s) > \mathtt{size}(s')$, this implies:

$$\mathit{spmax}(s) \;=\; \max(\mathtt{size}(s), \mathit{spmax}(s_1)) \;\geq\; \mathit{spmax}(s')$$

If diagram 2 is applicable, then $\mathit{spmax}(s_1) = \mathit{spmax}(s_2) \geq \mathit{spmax}(s_3)$ by Theorem 3.6 and Theorem 3.5. Proposition 3.6 shows $\mathit{spmax}(s_3) \geq \mathit{spmax}(s')$. Since also $\mathit{spmax}(s) \geq \mathit{spmax}(s_1)$, we obtain $\mathit{spmax}(s) \geq \mathit{spmax}(s')$.

In the case of diagram 3, $s \xrightarrow{LRPgc,a} s_1$ is the same reduction as $s \xrightarrow{\mathbb{T},caseId} s_1$, hence the claim holds obviously.

In the case of diagram 4, an application of the induction hypothesis to s_1 yields $\mathit{spmax}(s_1) \geq \mathit{spmax}(s_3)$. Since (gc) is a maximal space improvement (see Theorem 3.2), $\mathit{spmax}(s_3) \geq \mathit{spmax}(s'_1)$ holds. Since along LRPgc-reduction sequences, *spmax* is decreasing, $\mathit{spmax}(s') \geq \mathit{spmax}(s'_2) \geq \mathit{spmax}(s'_1)$ holds. Also $\mathtt{size}(s) \geq \mathtt{size}(s')$ holds, thus we obtain

$$\begin{aligned}\mathit{spmax}(s) &= \max(\mathtt{size}(s), \mathit{spmax}(s_1))\\ &\geq \max(\mathtt{size}(s'), \mathtt{size}(s'_2), \mathit{spmax}(s'_1)) = \mathit{spmax}(s')\end{aligned}$$

For diagram 5, the induction hypothesis can be applied to s_1 and the remaining computation is similar, since (gc) is a maximal space improvement.

For diagram 6, $\mathit{spmax}(s) \geq \mathit{spmax}(s_1) \geq \mathit{spmax}(s')$ holds because of Theorem 3.9 and Theorem 3.10 and the claim is proved.

Finally we apply Lemma 3.6 to show the claim.

3.4.12 Space Properties of (cse) and (soec)

In this section we show that (cse) and (soec) both introduce space leaks and thus are not space-safe. However it is easy to see that (soec) is a time improvement and (SSS17) shows that (cse) is a time improvement.

Proposition 3.7 ((cse) is a Space Leak)
The transformation (cse) is a space leak.

Proof
We reuse an example similar to the example in (BR00).

The expression s is given in a Haskell-like notation, using integers, but can also be defined in LRPgc:

$$\texttt{if}\ (\texttt{last}\ [1..n]) > 0\ \texttt{then}\ [1..n]\ \texttt{else}\ \texttt{Nil}$$

where $[1..n]$ is the expression that lazily generates a list $[1, \ldots, n]$ and `last` returns the last element of a list, i.e. forcing tail-strictness. Thus s evaluates the list expression until the last element is found and then evaluates the same expression again to $1 : [2..n]$. Due to eager garbage collection, the list elements generated by `last` $[1..n]$ are garbage collected directly after creation, only requiring constant space and the result list also only requires constant space. In LRPgc the evaluation will also generate `letrec`-environments, perhaps with long indirection chains. Our space measure ignores these, but a simple change in evaluation order where such indirections are shortened (see e.g. (DS16)) will also lead to a constant space on an abstract machine.

If we have $s \xrightarrow{cse} s'$, then s' is:

$$s' = \texttt{letrec}\ x = [1..n]\ \texttt{in}\ (\texttt{if}\ (\texttt{last}\ x) > 0\ \texttt{then}\ x\ \texttt{else}\ \texttt{Nil})$$

The evaluation of s' behaves different to s: It first evaluates the list and since it is needed later, it is stored in full length so that the second x only uses the already evaluated list.

The size required is a linear function in n. Seen from a complexity point of view, there is no real bound on this maximal space increase: The example can be adapted using any computable function f on n by modifying the list to $[1..f(n)]$.

Obviously this example is a space leak according to our definition of space leaks (Definition 3.8), where the reduction contexts contains the list definition.

Proposition 3.8 ((soec) is a Space Leak)
The transformation (soec) is a space leak.

Proof
Follows from the examples and arguments in (BR00).

3.4.13 Space Properties of (cp) and (cpS)

(cp) and (cpS) are both not maximal space improvements. We start with a complete set of forking diagrams for (cpS):

Lemma 3.15 (Forking Diagrams for (cpS))
A complete set of forking diagrams for (cpS) in surface contexts is as follows.

$$\begin{tikzcd}
s \arrow[r, "\mathbb{S},cpS"] \arrow[d, "LRPgc,a"'] & s' \arrow[d, dashed, "LRPgc,a"] \\
s_1 \arrow[r, dashed, "\mathbb{S},cpS"'] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{S},cpS"] \arrow[d, "LRPgc,a"'] & s' \arrow[dl, dashed, "LRPgc,a"] \\
s_1 &
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{S},cpS"] \arrow[d, "LRPgc,gc"'] & s' \arrow[ddl, dashed, "LRPgc,gc"] \\
s_1 \arrow[d, "LRPgc,cp"'] & \\
s_2 &
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[d, "LRPgc,cp"'] \arrow[d, dashed, bend left, "\mathbb{S},cpS"] \\
s'
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{S},cpS"] \arrow[dr, dashed, "\mathbb{S},ucp"'] \arrow[dd, "LRPgc,a"'] & s' \arrow[d, dashed, "LRPgc,gc"] \\
 & s'_2 \arrow[d, dashed, "LRPgc,a"] \\
s_1 \arrow[r, dashed, "\mathbb{S},ucp"'] & s'_1
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[rr, "\mathbb{S},cpS"] \arrow[dr, dashed, "\mathbb{S},ucp"] \arrow[dd, "LRPgc,gc"'] & & s' \arrow[dl, dashed, "gc"'] \arrow[dd, dashed, "LRPgc,gc"] \\
 & s''_2 \arrow[dr, dashed, "gc"'] & \\
s_1 \arrow[rr, dashed, "\mathbb{S},ucp"'] & & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{S},cpS"] \arrow[dr, dashed, "\mathbb{S},ucp"] \arrow[d, "LRPgc,a"'] & s' \arrow[d, dashed, "LRPgc,gc"] \\
s_1 \arrow[d, "LRPgc,gc"'] & s'_1 \arrow[dl, dashed, "LRPgc,a"] \\
s_2 &
\end{tikzcd}$$

$a \in \{seq, case\}$

Proof
We have to take into account that (cpS) may trigger a garbage collection, but only in the case that corresponds to an (ucp)-transformation. The case analysis is then straightforward.

We give an analysis of the maximal space increase of (cpS) in surface contexts:

Proposition 3.9 (Space-Property of (cpS) in Surface Contexts)
The transformation $(\mathbb{S}, cpS)$ increases the maximal space at most by $\mathtt{size}(v)$, where v is the copied abstraction.

Proof
Let $s \xrightarrow{\mathbb{S},cpS} s'$, where v is the copied abstraction.

We use the diagrams in Lemma 3.15 and prove the claim by induction on the following measure of s: (i) the number of *LCSC*-reductions μ_1, (ii) the measure μ_{lll} and (iii) the measure `synsize`.

If s is a LRPgc-WHNF, then s' is also a LRPgc-WHNF and the claim holds, since $\texttt{size}(s) = \texttt{size}(v) = \texttt{size}(s')$.

Now s has a LRPgc-reduction and we go through each applicable diagram in turn.

If diagram 1 applies, then the induction hypothesis applies to s_1 and we have $\texttt{size}(s) + \texttt{size}(v) = \texttt{size}(s')$, hence $\mathit{spmax}(s') \leq \mathit{spmax}(s) + \texttt{size}(v)$.

If diagram 2, 3 or 4 applies, then the computation is similar as above.

If diagram 5 applies, then $\mathit{spmax}(s) = \mathit{spmax}(s')$ and $\mathit{spmax}(s_1) = \mathit{spmax}(s_1')$ by Theorem 3.10. Then we have $\mathit{spmax}(s) = \max(\texttt{size}(s), \mathit{spmax}(s_1))$ and also $\mathit{spmax}(s') = \max(\texttt{size}(s'), \texttt{size}(s_2'), \mathit{spmax}(s_1))$. Hence the claim holds in this case. The reasoning for diagram 6 is almost the same.

If diagram 7 applies, we have $\mathit{spmax}(s) = \max(\texttt{size}(s), \mathit{spmax}(s_1))$ and also $\mathit{spmax}(s') = \max(\texttt{size}(s'), \texttt{size}(s_1'), \mathit{spmax}(s_2))$. Hence the claim holds.

We also give an estimation for the general (cp):

Theorem 3.12 (Space-Property of (cp))
If $s \xrightarrow{cp} t$, then $\mathit{spmax}(t) \leq (\mathrm{rln}(s) + 2) \cdot \texttt{size}(v) + \mathit{spmax}(s)$.

Proof
Since (cpS) is analyzed above, we assume that the target position of (cp) is within an abstraction.

Let p be the position of the variable x that is replaced by the (cp) transformation. This position is now labeled with L. We now analyze the trace of this label in a normal order reduction sequence $s \xrightarrow{LRPgc} s_1 \xrightarrow{LRPgc} \dots \xrightarrow{LRPgc} s_n$ where we assume that it may occur multiple times in the expressions and we also assume that the label is removed if it is replaced by an abstraction.

An invariant property is, that for every s_i the label L occurs at most once in every subexpression that is an abstraction. This can be verified by examining the effect of all possible normal order reduction steps, where we only consider the rules that make a change:

- (cp): It may copy an abstraction with a label in a reduction context of s_i.
- (lbeta): It may remove a lambda from the abstraction that contains a label L.

Since every (cp) applied in normal order is either the last reduction step in a successful normal order reduction sequence or followed by an (lbeta)-reduction, the total number of label occurrences is at most $\mathrm{rln}(s) + 2$.

Now we examine the normal order reduction sequence of t. The only difference is that the label positions are occupied by the copied abstraction. Hence the maximal size difference is $(\mathrm{rln}(s) + 2) \cdot \texttt{size}(v)$.

This also allows an estimation of several applications of $(\mathbb{S}, cpS)$:

Proposition 3.10 (Estimation of Space Increase of Several $(\mathbb{S}, cpS)$)
Let s be an expression. If s is transformed into s' by an arbitrary number of space improvements that do not increase the size of abstractions, including at most n transformations that increase the maximal space consumption by at most c_i for $i \in \{1, \dots, n\}$ and also by m transformations $(\mathbb{S}, cpS)$, then $\mathit{spmax}(s') \leq \mathit{spmax}(s) + (\sum c_i) + m \cdot V$, where V is the maximum of the sizes of abstractions in s.

Proof
This follows from Proposition 3.9 and since $\xrightarrow{(\mathbb{S}, cpS)}$ does not increase the size of abstractions.

The maximal space increase of (cp) may be linear in the number of reduction steps and exponentially in the number of applications of the (cp)-transformation. Examples for this behavior can be constructed as in the proof of Proposition 3.7.

3.4.14 Summary

In this section we summarize the results of the previous sections.

The normal order reduction is a maximal space improvement except of (cp):

Theorem 3.13 (Space-Property of Normal Order Reduction in LRP)
The normal order reduction of LRP is a maximal space improvement except of (cp).

Proof
This follows from Theorem 3.1, Theorem 3.3, Theorem 3.4 and Theorem 3.9.

We give an overview of all results w.r.t. space consumption of this section, where previous work (see (SSS17, SSS15a, SS15, SSS16b, SSS15b)) shows, that all of the following transformations are time improvements.

Theorem 3.14 (Space-Properties of Several LRP-Transformations)
The following table shows the space properties of all transformations analyzed in this section:

Space-Property	Transformations
$\leq_{\mathit{spmax}}$	(lbeta), (seq), (case), (lll), (gc), (case*), (caseId)
$\sim_{\mathit{spmax}}$	(cpx), (abs), (abse), (xch), (ucp), (case-cx), (gc=)
$\not\leq_{\mathit{spmax}}$	(cpcx), (cpS)
space-safe up to 1	($\mathbb{T}$,cpcxT)
space-safe up to `size`(v) where v is the copied abstraction	($\mathbb{S}$,cpS)
space-leak	(cp), (cse), (soec)

Proof
This follows from Theorem 3.2, Theorem 3.5, Theorem 3.6, Theorem 3.7, Theorem 3.8, Theorem 3.10, Theorem 3.11, Theorem 3.12, Theorem 3.13, Proposition 3.6, Proposition 3.7, Proposition 3.8 and Proposition 3.9.

3.5 Optimizations with Controlled Space Usage

Especially transformations like (cse), that are space-leaks but time improvements lead to the following proposal of space-controlled program optimizations in LRP:

1. Consider a program that is to be optimized by transformations.
2. Use only program transformations that are correct w.r.t. contextual equivalence.
3. Use only time improving transformations that are also space improvements. Several of such transformations are given by the table of Theorem 3.14. Since only space improvements are applied, the program itself is never enlarged.
4. Further time improving transformations can be applied that are not guaranteed to be space improvements, but there is an upper bound on the maximal space increase like ($\mathbb{T}$,cpcxT) and ($\mathbb{S}$,cpS).
5. There are transformations such as unrestricted (cp) and common subexpression elimination (cse), that are time improvements but as the concrete transformations may be space leaks, it comes with a higher risk of high space consumption. Hence in such cases further information on the maximal space increase is required for space-safe optimizations.

We now give examples for space improvements of recursive functions. See Section 2.6 for definitions of needed functions.

3.5.1 Associativity of Append

In (GS01), associativity of append was analyzed and the results use several variants of their improvement orderings. In particular their observation of stack and heap space made the analysis rather complex. We got an easier to obtain and to grasp result due to our relaxed measure of space.

For a precise definition of the append-function for lists see Section 2.6.

Proposition 3.11 (Space-Difference between Left- and Right-Associative Append)
Under the assumption that only the lists are evaluated, the following inequality holds, where xs, ys and zs are variables.

$$\mathit{spmax}(\mathbb{R}[((xs \mathbin{++} ys) \mathbin{++} zs)]) \leq 4 + \mathit{spmax}(\mathbb{R}[(xs \mathbin{++} (ys \mathbin{++} zs))])$$

Proof
We start with the same reduction context $\mathbb{R}$. Then $\mathbb{R}[((xs \mathbin{++} ys) \mathbin{++} zs)]$ has to be compared with $\mathbb{R}[(xs \mathbin{++} (ys \mathbin{++} zs))]$ and induction is used on the number of recursive expansions of ++. The `letrec`-environments are now shifted to the top using the space-properties of (lll) and (ucp) (see Theorem 3.3 and Theorem 3.10).

First we compute the left-hand side:

1. The body of append is copied: $\mathbb{R}[((xs \mathbin{++} ys) \mathbin{++}_{body} zs)]$

2. Two (lbeta) reduction steps lead to:

$$\mathbb{R}[\texttt{letrec}\ xs_1 = (xs \mathbin{++} ys), ys_1 = zs\ \texttt{in}\ (\texttt{case}\ xs_1\ \ldots)]$$

3. The body of append is copied:

$$\mathbb{R}[\texttt{letrec}\ xs_1 = (xs \mathbin{++}_{body} ys), ys_1 = zs\ \texttt{in}\ (\texttt{case}\ xs_1\ \ldots)]$$

4. Again two (lbeta)-steps are applied and we get:

$$\mathbb{R}[\texttt{letrec}\ xs_1 = (\texttt{case}\ xs_2\ \ldots), xs_2 = xs, ys_2 = ys, ys_1 = zs\ \texttt{in}\ (\texttt{case}\ xs_1\ \ldots)]$$

The computation for the right-hand side is as follows:

1. The body of append is copied:

$$\mathbb{R}[(xs \mathbin{++}_{body} (ys \mathbin{++} zs))]$$

2. Two (lbeta) reduction steps lead to:

$$\mathbb{R}[\texttt{letrec}\ xs_1 = xs, ys_1 = (ys \mathbin{++} zs)\ \texttt{in}\ (\texttt{case}\ xs_1\ \ldots))$$

Now we have the following cases:

1. The local evaluation of xs does not terminate. Then the space improvement relation holds independent of the size.
2. xs locally evaluates to the empty list. For the left-hand side the reduction result is as follows:

$$\mathbb{R}[\texttt{letrec}\ xs_1 = ys_2, xs_2 = xs, ys_2 = ys, ys_1 = zs\ \texttt{in}\ (\texttt{case}\ xs_1\ \ldots)]$$

   For the right-hand side we have $\mathbb{R}[\texttt{letrec}\ xs_1 = xs, ys_1 = (ys \mathbin{++} zs)\ \texttt{in}\ ys_1)]$, the next step yields $\mathbb{R}[\texttt{letrec}\ xs_1 = xs, ys_1 = (ys \mathbin{++}_{body} zs)\}\ \texttt{in}\ ys_1)]$ and then $\mathbb{R}[\texttt{letrec}\ xs_1 = xs, ys_1 = (\texttt{case}\ ys\ \texttt{of}\ \{(\texttt{Nil} \to zs)\ \ldots\})\ \texttt{in}\ ys_1]$, which is the same as for the left-hand side. The space maximum of the left expression is higher by 4 than the right one. The latter steps do not contribute to the space maximum, thus the claim holds in this case.
3. xs locally evaluates to $(a : as)$. For the left-hand side we first have:

$$\mathbb{R}[\texttt{letrec}\ xs_1 = a : (as \mathbin{++} ys), ys_1 = zs\}\ \texttt{in}\ (\texttt{case}\ xs_1\ \ldots)]$$

   Then this reduces to $\mathbb{R}[(a : ((as \mathbin{++} ys) \mathbin{++} zs))]$, where xs_1 is garbage collected. For the right-hand side we have $\mathbb{R}[a : (as \mathbin{++} (ys \mathbin{++} zs))]$. Now induction on the number of steps shows the claim.

Finally an application of Lemma 3.6 shows the claim.

Note that for the example in (GS01), where the difference in space consumption is linear, the hole of the context is within an abstraction.

3.5.2 Two Sum Variants

We reconsider the definitions of three variants of sum of a list and the space analysis in (GS01). If the list is unevaluated, then the different sum-versions below would change the evaluation order of parts of the expressions, which might introduce space leaks. Hence, we avoid this complication for our analysis and use a fully evaluated list $[1, \ldots, n]$ of integers as argument for the sum-functions.

We make further simplifications for the analysis: We assume that positive integers are available (for example Peano integers) and assume that an integer occupies a space unit of 1. Also it is assumed that addition (+) is a strict function in two arguments and that $n + m$ immediately returns the result without using extra space. These simplifications are justified, because we are only interested in analyzing the recursive variant in comparison with the tail recursive variants and since we could also use Boolean values or constants for numbers.

The different sum-variants are defined as follows:

$$\begin{array}{lcl}
\texttt{sum} & = & \lambda xs.\texttt{case}\ xs\ \texttt{of}\ \{(\texttt{Nil} \to 0)\ ((y : ys) \to y + (\texttt{sum}\ ys))\} \\
\texttt{sum}' & = & \lambda xs.\texttt{asum}\ 0\ xs \\
\texttt{asum} & = & \lambda a.\lambda xs.\texttt{case}\ xs\ \texttt{of}\ \{(\texttt{Nil} \to a)\ ((y : ys) \to \texttt{asum}\ (a + y)\ ys)\} \\
\texttt{sum}'' & = & \lambda xs.\texttt{asum}'\ 0\ xs \\
\texttt{asum}' & = & \lambda a.\lambda xs.\texttt{case}\ xs\ \texttt{of}\ \{ \\
 & & \qquad (\texttt{Nil} \to a) \\
 & & \qquad ((y : ys) \to \texttt{letrec}\ a' = a + y\ \texttt{in}\ \texttt{seq}\ a'\ (\texttt{asum}'\ a'\ ys))\}
\end{array}$$

Let us assume that the definitions of the functions are in an outer letrec-environment and then we compare sum $[1, \ldots, n]$, sum′ $[1, \ldots, n]$ and sum″ $[1, \ldots, n]$. We also assume that the input environment including the list is not garbage collected during the evaluation.

First we analyze the space usage for an empty list:

- For sum the maximal space consumption, without the outer letrec and without the input list, is: $8 + 1 = 9$

$$\begin{array}{ll}
 & \texttt{sum}\ \texttt{Nil} \\
\to & \texttt{sum}_{body}\ \texttt{Nil} \\
\to & \texttt{letrec}\ xs = \texttt{Nil}\ \texttt{in}\ \texttt{case}\ xs\ \texttt{of}\ \{(\texttt{Nil} \to\ 0)\ ((y : ys) \to\ y + \texttt{sum}\ ys)\} \\
\to & 0
\end{array}$$

- For sum′ the maximal space consumption, without the outer letrec and without the input list, is: $9 + 1 = 10$

$$\begin{array}{ll}
 & \texttt{sum}'\ \texttt{Nil} \\
\to & \texttt{sum}'_{body}\ \texttt{Nil} \\
\to & \texttt{letrec}\ xs = \texttt{Nil}\ \texttt{in}\ \texttt{asum}\ 0\ xs \\
\to & \texttt{letrec}\ xs = \texttt{Nil}\ \texttt{in}\ \texttt{asum}_{body}\ 0\ xs \\
\to & \texttt{letrec}\ xs = \texttt{Nil}, a = 0, xs' = xs\ \texttt{in}\ \texttt{case}\ xs'\ \texttt{of}\ \{(\texttt{Nil} \to a) \\
 & \qquad\qquad\qquad\qquad ((y{:}ys) \to \texttt{asum}\ (y{+}a)\ ys)\}
\end{array}$$

- For sum″ the analysis is analogous, the maximal space consumption is $10 + 1 = 11$.

If the lists are not empty, then the analysis results in intermediate steps:

- For $\mathtt{sum}\ [i,\ldots,n]$ the maximal space consumption, without the outer `letrec` and without the input list, is: $2\cdot n+2+\mathtt{size}(\mathtt{sum}_{body}) = 2\cdot(n+1)+8$
  The corresponding expression:

  $$1+(2+\ldots+(n+((\lambda xs.\mathtt{case}\ xs\ \mathtt{of}\ \{(\mathtt{Nil}\rightarrow\ 0)\ ((y:ys)\rightarrow\ y+\mathtt{sum}\ ys)\})\ \mathtt{Nil})))$$

- For $\mathtt{sum}'\ [i,\ldots,n]$ the maximal space consumption, without the outer `letrec` and without the input list, is: $2\cdot n+2+\mathtt{size}(\mathtt{asum}_{body}) = 2\cdot(n+1)+9$
  The corresponding expression:

  $$\mathtt{letrec}\ xs_n = \mathtt{Nil}\ \mathtt{in}\ \mathtt{asum}_{body}\ (((0+1)+2)+\ldots+n)\ xs_n$$

- For $\mathtt{sum}''\ [i,\ldots,n]$ the maximal space consumption, without the outer `letrec` and without the input list, is the following constant: $1+2+10=13$
  The corresponding expression:

  $$\mathtt{letrec}\ ys = \mathtt{Nil}, a' = n\ \mathtt{in}\ \mathtt{seq}\ a'\ (\mathtt{asum}'_{body}\ a'\ ys)$$

The functions `sum`, `sum′` and `sum″` are not related by any improvement relation due to the change in the evaluation order of the spine and elements of the argument list, in case the list is not completely evaluated. In the latter case transforming one into the other may indeed be a space-leak, independent of the length of the list since it would be an instance of (soec).

3.5.3 Weak Space Improvements

As a further comparison we check and compare our results with those for weak space improvements in Figure 2 in (GS99):

The claim on (weak-value-beta) there appears to be practically almost useless: Copying once indeed can only increase the space by a linear function in the size of the program, even copying into an abstraction is permitted. However, repeating (weak-value-beta) n-times may increase the program exponentially in n by repeated doubling. The transformation rule in (GS99) permits the following, where $\mathbb{C}$ is a value as context.

$$\begin{array}{ll} & \mathtt{letrec}\ x = \mathbb{C}[x]\ \mathtt{in}\ \mathbb{C}[x] \\ \rightarrow & \mathtt{letrec}\ x = \mathbb{C}[\mathbb{C}[x]]\ \mathtt{in}\ \mathbb{C}[\mathbb{C}[x]] \\ \rightarrow & \mathtt{letrec}\ x = \mathbb{C}[\mathbb{C}[\mathbb{C}[\mathbb{C}[x]]]]\ \mathtt{in}\ \mathbb{C}[\mathbb{C}[\mathbb{C}[\mathbb{C}[x]]]] \end{array}$$

Hence a sequence of several weak space improvement steps is not space-safe in the intuitive sense. According to our definition it is a space leak for this particular example.

Our foundation allows to improve the claims on the space-properties of last two let-shuffling rules of (GS99), which are (strong) space improvements w.r.t. our measure and definitions, since we have proved that (lll) is a space equivalence.

3.5.4 Evaluations in Limited Space

A practical approach is to limit the allowed space usage by a function.

Definition 3.13 (Evaluation Sequence)
Let S be a set of closed expressions of the same datatype, $s \in S$ and the full evaluation result is *eval*(s).
Let an *evaluation sequence* of s be a list L of all positions of *eval*(s), such that whenever p_1 is a (proper) prefix of p_2, then p_1 is earlier in L than p_2.
An evaluation of s *controlled by* L is the reduction sequence for s where the lazy evaluation evaluates the tail-nodes top-down in s, starting with the first position in L and whenever the evaluation stops, tries the expression at the next position in L.

Note that the expression s itself may evaluate deeper, for example:

$$\texttt{letrec}\ xs = [1..n]\ \texttt{in seq}\ (\texttt{length}\ xs)\ xs$$

Now we can define the notion of limited space:

Definition 3.14 (Evaluation in Limited Space)
If there is a function f from integers to integers, such that for all $s \in S$ and all evaluation sequences L_s of s, the evaluation of s controlled by L_s requires at most space $\texttt{size}(\textit{eval}(s)) + f(\texttt{size}(\textit{eval}(s)))$, then we say the expressions of S can be evaluated in f-space. If $f(x)$ is a constant, then it can be evaluated in constant space.

For example, the set of list expressions $[1..n]$ can be evaluated in constant space. We also conclude that for all list expressions that result in finite lists of integers and that can be evaluated in constant space and are non-empty, replacing `sum` by `sum''` (see Section 3.5.2) is a space improvement.

3.5.5 Examples for the Practical Approach

As conclusion of this section we give examples of the practical approach of optimizing programs w.r.t. space consumption.

We reconsider the (caseId)-transformation. The transformations in the following all require typing, they are not correct in an untyped setting and (caseId) is the heart of these transformations.

Space-Comparison of map $(\lambda x.x)$ and id

To show that `id` is a maximal space improvement of `map` $(\lambda x.x)$, we first show the contextual equivalence of the expressions.

The applicative bisimulation in the call-by-name variant of the calculus LRP can be used here (see (SS14)) together with the equivalence of LRP and LRPgc (see Proposition 3.1). The proof is by standard methods. We have to check the cases:

1. If xs is $\bot$, then the equivalence holds.
2. If xs = `Nil`, then the equivalence holds by simple computation.

3. If $xs = (a : as)$, then $\texttt{map}\ (\lambda x.x)\ (a : as)$ reduces to $(a : \texttt{map}\ (\lambda x.x)\ as)$ and the right-hand side to $(a : as)$. We now can go on with the applicative bisimulation.

To show that the space improvement property holds, we use the context lemma for maximal space improvements (Lemma 3.6). Let $\mathbb{R}$ be a reduction context. We compare $\mathbb{R}[\texttt{map}\ (\lambda x.x)\ xs]$ and $\mathbb{R}[((\lambda x.x)\ xs)]$. Then we have the following cases:

1. If xs is not convergent, then the space improvement property holds.
2. If the left-hand side reduces to $\mathbb{R}[\texttt{map}_{body}\ (\lambda x.x)\ xs]$, then this reduces to:

   $$\mathbb{R}[\texttt{letrec}\ f = (\lambda x.x), xs' = xs\ \texttt{in case}\ xs' \ldots]$$

   If xs reduces to the empty list, then the maximal space consumption of the right-hand side is smaller than that of the left-hand side. If xs reduces to $(a : as)$, then we also see that the maximal space consumption of the left-hand side is greater than that of the right-hand side and by induction, the maximal space consumption of the left-hand side is greater then that of the right-hand side.

Space-Comparison of `foldr` (:) [] and `id`

To show that `id` is a maximal space improvement of `foldr` (:) [], we first show the contextual equivalence of the expressions.

As in the example before we use applicative bisimulation in the call-by-name variant of LRP, where we have to check the following cases:

1. If xs is $\perp$, then the equivalence holds, since the functions are strict in the argument.
2. If $xs = \texttt{Nil}$, then the equivalence holds by simple computation.
3. If $xs = (a : as)$, then $\texttt{foldr}\ (:)\ []\ (a : as)$ reduces to $(a : \texttt{foldr}\ (:)\ []\ as)$ and the right-hand side to $(a : as)$. This is sufficient for the applicative bisimulation.

To show that the space improvement property holds, we use the context lemma for maximal space improvements (Lemma 3.6). Let $\mathbb{R}$ be a reduction context. We compare $\mathbb{R}[\texttt{foldr}\ (:)\ []\ xs]$ and $\mathbb{R}[((\lambda x.x)\ xs)]$. Then we have the following cases:

1. If xs is not convergent, then the space improvement property holds.
2. If the left-hand side reduces to $\mathbb{R}[\texttt{foldr}_{body}\ (:)\ []\ xs]$, then this reduces to:

   $$\mathbb{R}[\texttt{letrec}\ f = (:), e = [], xs' = xs\ \texttt{in case}\ xs' \ldots]$$

   If xs reduces to the empty list, then the maximal space consumption of the right-hand side is smaller than that of the left-hand side. If xs reduces to $(a : as)$, then we also see that the maximal space consumption of the left-hand side is greater than that of the right-hand side and by induction, the maximal space consumption of the left-hand side is greater then that of the right-hand side.

Space-Comparison of `filter` $(\lambda x.\texttt{True})$ and `id`

To show that `id` is a maximal space improvement of `filter` $(\lambda x.\texttt{True})$, we first show the contextual equivalence of the expressions.

As in the examples before we use applicative bisimulation in the call-by-name variant of LRP, where we have to check the following cases:

1. If xs is $\perp$, then the equivalence holds, since the functions are strict in the argument.
2. If $xs = \texttt{Nil}$, then the equivalence holds by simple computation.
3. If $xs = (a{:}as)$, then `filter` $(\lambda x.\texttt{True})$ $(a{:}as)$ reduces to $a : \texttt{filter}\ (\lambda x.\texttt{True})\ as$ and the right-hand side to $(a{:}as)$. This is sufficient for the applicative bisimulation.

To show that the space improvement property holds, we use the context lemma for maximal space improvements (Lemma 3.6). Let $\mathbb{R}$ be a reduction context. We compare $\mathbb{R}[\texttt{filter}\ (\lambda x.\texttt{True})\ xs]$ and $\mathbb{R}[((\lambda x.x)\ xs)]$. Then we have the following cases:

1. If xs is not convergent, then the space improvement property holds.
2. If the left-hand side reduces to $\mathbb{R}[\texttt{filter}_{body}\ (\lambda x.\texttt{True})\ xs]$, then this reduces to:

$$\mathbb{R}[\texttt{letrec}\ p = (\lambda x.\texttt{True}), xs' = xs\ \texttt{in case}\ xs' \ldots]$$

   If xs reduces to the empty list, then the maximal space consumption of the right-hand side is smaller than that of the left-hand side. If xs reduces to $(a : as)$, then we also see that the maximal space consumption of the left-hand side is greater than that of the right-hand side and by induction, the maximal space consumption of the left-hand side is greater then that of the right-hand side.

3.6 Environment for Space Analyses

In this section we construct an environment for space analyses. This environment helps to falsify conjectures that certain transformations are space improvement and moreover we give the results of different analyses, including the practical impact of garbage collection.

3.6.1 Abstract Machine M1$_{sp}$

We adapt the abstract machine M1 as defined in Section 2.4.2 so that it has a space consumption that is compatible to the calculus LRPgc and therefore can be used to construct an environment to perform space analyses for LRPgc. The result is the abstract machine M1$_{sp}$. This section is primarily based on (DS16) and parts of the correctness are from (SSD18).

First of all we give a definition of a garbage collector as an additional transition rule for the abstract machine M1, that has to be applied whenever possible to be compatible to the eager garbage collection approach of LRPgc (compare Definition 3.2):

Definition 3.15 (Additional M1-Rule (GC))
(GC) $\langle\mathcal{H}, \{x_i = s_i\} \mid s \mid \mathcal{S}\rangle \to \langle\mathcal{H} \mid s \mid \mathcal{S}\rangle$
where $\{x_i = s_i\}$ is the maximal set such that for all i:
$x_i \notin \mathit{FV}(\mathcal{H}), x_i \notin \mathit{FV}(s), \#\mathrm{app}(x_i) \notin \mathcal{S}, \#\mathrm{seq}(x_i) \notin \mathcal{S}$ and if $x_i \in \mathit{FV}(alts)$ then $\#\mathrm{case}(alts) \notin \mathcal{S}$

Since the heap can be seen as a top-`letrec` and (GC) only removes bindings on the heap, this is a direct implementation of (gc) in LRPgc (compare Definition 3.2). The requirements for the stack are needed, because the M1 stores parts of the program on the stack during execution.

The space measure *spmax* (see Definition 3.6) abstracts over local peaks, since only the maximum of such values that do not allow a garbage collection are considered during the maximum. This was not only very useful during the last sections, but also w.r.t. the abstract machine. For example: Let s be the following LRPgc-expression, where t is an arbitrary LRPgc-expression:

$$((\texttt{seq}\ \texttt{True}\ (\lambda x.t))\ \texttt{True})$$

Then we can evaluate s using the normal order reduction of LRPgc, where we denote the `size` of each expression besides:

$$\begin{array}{rl|l}
 & ((\texttt{seq}\ \texttt{True}\ (\lambda x.t))\ \texttt{True}) & 5 + \texttt{size}(t) \\
\xrightarrow{LRPgc,seq-c} & ((\lambda x.t)\ \texttt{True}) & 3 + \texttt{size}(t) \\
\xrightarrow{LRPgc,lbeta} & (\texttt{letrec}\ x = \texttt{True}\ \texttt{in}\ t) & 1 + \texttt{size}(t) \\
\dots & &
\end{array}$$

The following machine expression s_{me} corresponds to s (see Definition 2.30 for the translation):

$$\texttt{letrec}\ x_1 = \texttt{True}, x_2 = \lambda x.t\ \texttt{in}\ ((\texttt{seq}\ \texttt{True}\ x_2)\ x_1)$$

We evaluate this expression s_{me} using the normal order reduction of LRPgc, where we again denote the `size` of each expression besides:

$$\begin{array}{rl|l}
 & \texttt{letrec}\ x_1 = \texttt{True}, x_2 = \lambda x.t\ \texttt{in}\ ((\texttt{seq}\ \texttt{True}\ x_2)\ x_1) & 5 + \texttt{size}(t) \\
\xrightarrow{LRPgc,seq-in} & \texttt{letrec}\ x_1 = \texttt{True}, x_2 = \lambda x.t\ \texttt{in}\ (x_2\ x_1) & 3 + \texttt{size}(t) \\
\xrightarrow{LRPgc,cp-in} & \texttt{letrec}\ x_1 = \texttt{True}, x_2 = \lambda x.t\ \texttt{in}\ ((\lambda y.t)\ x_1) & 4 + 2 \cdot \texttt{size}(t) \\
\xrightarrow{LRPgc,gc} & \texttt{letrec}\ x_1 = \texttt{True}\ \texttt{in}\ ((\lambda y.t)\ x_1) & 3 + \texttt{size}(t) \\
\xrightarrow{LRPgc,lbeta} & \texttt{letrec}\ x_1 = \texttt{True}\ \texttt{in}\ (\texttt{letrec}\ y = x_1\ \texttt{in}\ t) & 1 + \texttt{size}(t) \\
\xrightarrow{LRPgc,llet-in} & \texttt{letrec}\ x_1 = \texttt{True}, y = x_1\ \texttt{in}\ t & 1 + \texttt{size}(t) \\
\dots & &
\end{array}$$

Since *spmax* only takes into account those expressions after the garbage collection, we have $\mathit{spmax}(s) = 5 + \texttt{size}(t) = \mathit{spmax}(s_{me})$, otherwise the duplicated abstraction $\lambda x.t$, that is directly garbage, would be counted and the *spmax* of s_{me} would be strictly greater than the one of s. Especially those kind of local space peaks distort the results and therefore are ignored by *spmax*.

However these kind of peaks can occur in other cases, where we first give definitions of the size of abstract machine states and the overall space measure for the abstract machine:

Definition 3.16 (Size of M1-State `msize`)
The size of a M1 state $S = \langle \mathcal{H} \mid s \mid \mathcal{S} \rangle$ is defined as follows:
Let $\mathcal{H} = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$, then the size of the heap $\texttt{msize}_{\mathcal{H}}$ is defined as $\sum_{i=1}^{n} \texttt{size}(e_i)$.
The size of stack $\texttt{msize}_{\mathcal{S}}$ is the sum of the sizes of the stack-entries: #app(x) and #seq(x) are counted as 1, #upd(x) as 0 and #case($alts$) as follows:
For a `case`-alternative $(c\, y_1\, \ldots\, y_n) \to t$ the size is defined as $1 + \texttt{size}(t)$ and the size of #case($alts$) is the sum over all of such sizes of the `case`-alternatives $alts$.
Finally $\texttt{msize}(S) := \texttt{msize}_{\mathcal{H}} + \texttt{size}(s) + \texttt{msize}_{\mathcal{S}}$.

`msize` is compatible to `size`: The heap of a state can be seen as an outer-`letrec`, hence it is compatible to `size`, that the mapping-variables of the heap are not counted, while the bindings are counted using `size`. The machine expression is directly measured by `size`. #app(x) and #seq(x) are measured with size 1, since the corresponding application and `seq`-expression is still not evaluated, even if it is not visible anymore as part of the control expression. #case($alts$) contains the unevaluated `case`-alternatives and #upd(x) is measured with size 0, since the (Lookup) that created this stack-entry only moved the corresponding heap-binding to the control expression. However after an (Update) the `msize` of a state might be increased, as we see in the following example:

Consider the LRPgc-expression t, where x occurs in the LRPgc-expression t':

$$\texttt{letrec}\ x = \texttt{True}, y = t'\ \texttt{in}\ \texttt{seq}\ x\ y$$

The LRPgc-evaluation only needs a single (seq-in)-step before the evaluation of t' starts:

$$\texttt{letrec}\ x = \texttt{True}, y = t'\ \texttt{in}\ y$$

Hence the maximal `msize` so far is $2 + \texttt{size}(t')$. Now we let the abstract machine M1 evaluate the machine expression t to the same point:

$$\begin{array}{rl|l}
 & \langle \varnothing \mid \texttt{letrec}\ x = \texttt{True}, y = t'\ \texttt{in}\ \texttt{seq}\ x\ y \mid [] \rangle & 2 + \texttt{size}(t') \\
\xrightarrow{Letrec} & \langle x = \texttt{True}, y = t' \mid \texttt{seq}\ x\ y \mid [] \rangle & 2 + \texttt{size}(t') \\
\xrightarrow{Unwind2} & \langle x = \texttt{True}, y = t' \mid x \mid [\#\text{seq}(y)] \rangle & 2 + \texttt{size}(t') \\
\xrightarrow{Lookup} & \langle y = t' \mid \texttt{True} \mid [\#\text{upd}(x), \#\text{seq}(y)] \rangle & 2 + \texttt{size}(t') \\
\xrightarrow{Update} & \langle x = \texttt{True}, y = t' \mid \texttt{True} \mid [\#\text{seq}(y)] \rangle & 3 + \texttt{size}(t') \\
\xrightarrow{Seq} & \langle x = \texttt{True}, y = t' \mid y \mid [] \rangle & 1 + \texttt{size}(t') \\
\ldots & &
\end{array}$$

The last state above can be directly translated to $\texttt{letrec}\ x = \texttt{True}, y = t'\ \texttt{in}\ y$ as LRPgc-expression. The abstract machine has $3 + \texttt{size}(t')$ as maximal `msize` so far, while LRPgc above only needed a maximum of $2 + \texttt{size}(t')$ to the same evaluation point.

This is caused by the (Update) before (Seq) can be performed and leads to differences in space measurement between LRPgc and M1, since the abstract machine M1 duplicates the entry, while a (seq-in) evaluates directly. From an implementation point of view a solution is the definition of an additional rule (UpdateSeq), that combines (Update) and the directly following (Seq) in such cases. Such an effect occurs whenever an (Update) is caused by a constructor applications. This leads to the following definition of the overall space measurement of evaluation sequences of the M1:

Definition 3.17 (M1 Space Measure *mspmax*)
Let s be a terminating machine expression, i.e. $\langle \varnothing \mid s \mid [] \rangle = S_1 \to \cdots \to S_n$. Then the space measure for the M1 is defined as follows:

$$\mathit{mspmax}(s) := \max\{\texttt{msize}(S_i) \mid 1 \le i \le n, \neg(S_{i-1} = \langle \mathcal{H}, c\ \vec{x}, \mathcal{S}\rangle \wedge S_{i-1} \xrightarrow{\mathit{Update}} S_i) \text{ and (GC) is not applicable to } S_i\}$$

I.e. states after (Update)-transitions are ignored for constructor applications and also states where (GC) is applicable.

Moreover every (Lookup) triggers an (Update). There are situations where a variable as control expression leads to another variable as control expression (e.g. variable chains in `letrec`-environments). For example the following state leads to three (Update) in sequence:

$$\langle \mathcal{H} \mid \texttt{True} \mid \#\mathrm{upd}(x) : \#\mathrm{upd}(y) : \#\mathrm{upd}(z) : \mathcal{S}\rangle$$

Seen as a `letrec`-environment, `letrec` $x = y, y = z, z =$ `True` after a few steps yields `letrec` $x =$ `True`$, y =$ `True`$, z =$ `True`. But LRP does copy such values right to the needed position, without copying it to each position of the corresponding chain (see Definition 2.5).

The following example even shows that the difference in space consumption is at least $c \cdot n$, where c is the size of the value v:

$$\texttt{letrec}\ id = (\lambda x.x), x_1 = (id\ x_2), \ldots, x_{n-1} = (id\ x_n), x_n = v\ \texttt{in}\ \texttt{seq}\ x_1\ (\texttt{T}\ x_1\ x_2\ \ldots\ x_n)$$

The tuple $(\texttt{T}\ x_1\ x_2\ \ldots\ x_n)$ ensures that none of the bindings can be removed by the garbage collector. The execution by the M1 leads to a sequence of n (Update)-transitions, where the value v gets copied to each binding of the chain.

To avoid this effect, the rule (SCRem) is introduced and has to be applied whenever possible:

Definition 3.18 (Additional M1-Rule (SCRem))
(SCRem) $\langle \mathcal{H} \mid s \mid \#\mathrm{upd}(x) : \#\mathrm{upd}(y) : \mathcal{S}\rangle \to \langle \mathcal{H}[x/y] \mid s[x/y] \mid \#\mathrm{upd}(x) : \mathcal{S}[x/y]\rangle$

If we consider the example above, then we have:

$$\begin{array}{ll} & \langle \mathcal{H} \mid \texttt{True} \mid \#\mathrm{upd}(x) : \#\mathrm{upd}(y) : \#\mathrm{upd}(z) : \mathcal{S}\rangle \\ \xrightarrow{\text{(SCRem)}} & \langle \mathcal{H}[x/y] \mid \texttt{True} \mid \#\mathrm{upd}(x) : \#\mathrm{upd}(z) : \mathcal{S}[x/y]\rangle \\ \xrightarrow{\text{(SCRem)}} & \langle \mathcal{H}[x/y, x/z] \mid \texttt{True} \mid \#\mathrm{upd}(x) : \mathcal{S}[x/y, x/z]\rangle \end{array}$$

The rule (SCRem) is correct, since the state $\langle \mathcal{H} \mid v \mid \#\text{upd}(x) : \#\text{upd}(y)\rangle$ corresponds to `letrec` $\mathcal{H}, x = v, y = x$ `in` y with $x \not\equiv y$ before application and after the application it is $\langle \mathcal{H}[x/y] \mid v[x/y] \mid \#\text{upd}(x)\rangle$ corresponding to `letrec` $\mathcal{H}[x/y], x = v[x/y]$ `in` $y[x/y]$ and replacing variables by variables is shown to be correct in (SSS08b).

Indirections in general lead to such effects, thus we moreover require that the machine expressions are free of variable-to-variable bindings in `letrec`-expressions. This can be achieved by straightforward substitutions before the evaluation starts.

Now we summarize the requirements and modifications made to the M1:

Definition 3.19 (Abstract Machine M1$_{sp}$)
The abstract machine M1$_{sp}$ is the abstract machine M1, but additionally the transition rules (GC) and (SCRem) have to be applied whenever possible and moreover an input machine expression is required to have no variable-to-variable bindings in `letrec`-expressions.

All our modifications and requirements yield a compatible space measurement between LRPgc and M1$_{sp}$:

Theorem 3.15 (Space Equivalence between LRPgc and M1$_{sp}$)
If an LRPgc-expression s is translated to its corresponding machine expression s_{me}, then s and s_{me} are space equivalent, if M1$_{sp}$ is used for the evaluation of s_{me}.

Proof
The reverse of the transformation (ucp) allows a step-by-step translation from LRPgc-expressions to machine expressions, where (ucp) is a space equivalence (see Theorem 3.10).

Since the garbage collection is similar and as often applied as the one of LRPgc, the only difference w.r.t. to evaluation and therefore space consumption are (Update)-transitions for constructors and indirections.

The (Update)-transitions for constructors are handled by not counting the intermediate expressions caused by (Update) if the control expression is a constructor (also a combination in form of additional rules such as (UpdateSeq) or (UpdateBranch) would solve this issue, that simply combine (Update) and (Seq) or (Update) and (Branch) in such cases).

Explicit variable-to-variable bindings are eliminated before the evaluation starts by straightforward substitutions. Implicit variable-to-variable-bindings may occur through the stack and are handled by the transition rule (SCRem), that has to be applied as often as possible.

Thus the abstract machine M1$_{sp}$ can be used to perform space analyses of LRPgc. This is useful to refute wrong space improvement conjectures and to perform analyses on larger and more complex programs. The M1$_{sp}$ is implemented in the tool LRPi together with an interface to define (space) measures, different garbage collection modes and other features, e.g. to execute an LRP-program with different inputs (a specified variable that is instantiated with a value). More details are in (DS16, Dal16).

3.6.2 Analyses of Examples

In this section we perform analyses using the tool LRPi, where the compatibility of its abstract machine $M1_{sp}$ was shown in Section 3.6.1.

The LRPi support conjectures of space improvements by affirmative tests or refutes the space improvement property of a transformation by finding a counterexample. Since LRPi only tests in the empty environment, a complete test would require to perform the test also within contexts, which cannot be done completely, since there are infinitely many, even using context lemmas to minimize the set of necessary contexts. Using a simulation mode, the contexts could be restricted to testing the functions on arguments. For these tests typing makes a big difference, since certain transformations are correct only if typing is respected and also the space improvement property may depend on the restriction to typed arguments or type-correct insertion into contexts.

In the following subsections we perform analyses using the LRPi. Needed function definitions can be found in Section 2.6. The LRPi can be found at:

`http://www.ki.cs.uni-frankfurt.de/research/lrpi`

Variants of `fold`

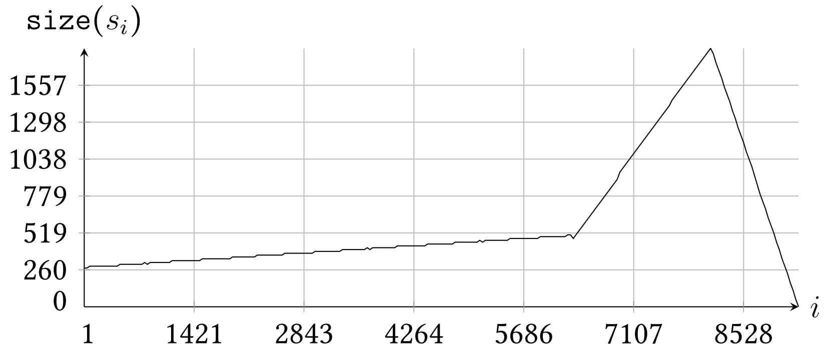
Following (SSS15a), we use the LRPi to find an example in which `foldl` is worse than `foldr` if preconditions on arguments are not fulfilled: Choosing `xor` for f and `False` as e, the precondition $f\ e\ s \leq f\ s\ e$ holds, but $(f\ (f\ s_1\ s_2)\ s_3) \leq (f\ s_1\ (f\ s_2\ s_3))$ is not fulfilled for s_1 = `True`, s_2 = `False`, s_3 = `False`. A list starting with a single `True` element followed by $k-1$ `False`-elements generated using a take-function/list-generator approach (using a Peano encoding to represent the numbers) is used as input list. We use `False` as neutral element.

Note that the Peano encoding has an impact on the `size`, since it consists of the constructors `Zero` and `Succ`, each with `size` 1. We configure LRPi to collect garbage whenever possible to be compatible to LRPgc and get the following results for the different variants of fold:

k	25	50	75	100	125	150	175	200	225
	`foldl` using `xor`								
mln	302	602	902	1202	1502	1802	2102	2402	2702
mlnall	1085	2160	3235	4310	5385	6460	7535	8610	9685
spmax	211	411	611	811	1011	1211	1411	1611	1811
	`foldl'` using `xor`								
mln	327	652	977	1302	1627	1952	2277	2602	2927
mlnall	1235	2460	3685	4910	6135	7360	8585	9810	11035
spmax	87	112	137	162	187	212	237	262	287
	`foldr` using `xor`								
mln	279	554	829	1104	1379	1654	1929	2204	2479
mlnall	1016	2016	3016	4016	5016	6016	7016	8016	9016
spmax	90	115	140	165	190	215	240	265	290

As we see, `foldr` indeed has a better runtime behavior than `foldl` and the space consumption of `foldr` and `foldl'` are almost equal. Moreover, we see that `foldl` has a much worse space behavior than `foldl'`. This difference is caused by the known stack problems of `foldl` that can be solved in the case of `xor` by using `foldl'` instead. The linear increase of the space consumption for `foldl'` and `foldr` is caused by the Peano numbers of the take-function/list-generator approach.

We can identify the stack overflow problem of the left-fold in this scenario in the following space diagram using $k = 225$, directly calculated by LRPi. Note that (gc)-reductions are not counted by mlnall, but counted in the following diagram:



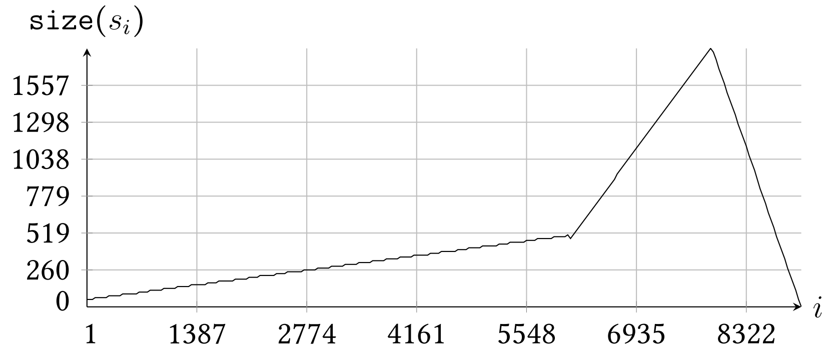
Let s_i be the i-th expression during execution. Because of lazy evaluation, the `foldl`-expression is expanded step by step without calculating any intermediate results until `foldl` itself is no longer required and removed by the garbage collector. This leaves a long chain of nested `(++)`-function calls that lead to the big rise of the curve, since this causes a long chain of (lbeta)- and (case)-transformations. The small decrease before the rise of the curve is caused by the removal of the definition of `foldl` by the garbage collector, because the definition of `foldl` is not needed anymore after the expansion is completed.

Moreover we want to analyze the impact of the Peano encoding. I.e. we perform the same analysis again, but the LRPi is configured to treat arbitrary Peano numbers as of `size` 1. The runtime results are the same as above, thus omitted:

k	25	50	75	100	125	150	175	200	225
spmax for `foldl`	211	411	611	811	1011	1211	1411	1611	1811
spmax for `foldl'`	63	63	63	63	63	63	63	63	63
spmax for `foldr`	66	66	66	66	66	66	66	66	66

The space consumption of `foldl'` and `foldr` is constant since the eager garbage collection can collect any so far processed parts of the list.

But even with the more realistic measurement of numbers, `foldl` has the same space behavior as with a fully counted Peano encoding as above, since the impact of the stack overflow problem of left-fold is greater then the space impact of the Peano numbers. This can be seen if we compare the following diagram for left-fold with $k = 225$ with the one above: The only difference is the weaker impact of the Peano numbers in the beginning. The diagram has a few steps less, since all data points where (GC) does not have an impact on the `size` are removed.



Now we analyze the impact of inlining w.r.t. fold-functions. We use the same input list and neutral element as above and motivated by the example above, the LRPi is again configured to treat the `size` of numbers as 1. First of all we give the definitions of the inlined variants of folds with `xor` as function (see Section 2.6 for the definitions of the fold-functions and `xor`):

Definition 3.20 (Inlined Versions of `foldl`, `foldl'` and `foldr` with `xor` as Function) We give the definition of the inlined variant of `foldl` with `xor` in detail:

```
foldlxor = λz, xs.case xs of {
                      ([] → z)
                      ((y : ys) → foldlxor
                                     (case z of {
                                        (True → case y of {
                                                   (True → False)
                                                   (False → True)})
                                        (False → y)})
                                     ys)}
```

The inlined versions `foldl'xor` and `foldrxor` can be constructed analogous to the inlined version of `foldl` above.

Inlining copies the defining lambda-expression for `xor` to a call site and then applies (lbeta), (ucp), (cpx), (gc) perhaps several times to obtain a non-optimized variant of inlined definitions. To obtain the definitions as given in Definition 3.20, the passing through of the unneeded combination function, that is now directly implemented in the body of the fold-function, is removed by the elimination of the corresponding argument.

k	100	200	300	400	500	600	700	800	900
	`foldl` using `xor`								(without inlining)
mln	1214	2414	3614	4814	6014	7214	8414	9614	10814
mlnall	4353	8653	12953	17253	21553	25853	30153	34453	38753
spmax	819	1619	2419	3219	4019	4819	5619	6419	7219
	`foldlxor`								(with inlining)
mln	910	1810	2710	3610	4510	5410	6310	7210	8110
mlnall	3543	7043	10543	14043	17543	21043	24543	28043	31543
spmax	869	1669	2469	3269	4069	4869	5669	6469	7269

k	100	200	300	400	500	600	700	800	900
	`foldl'` using `xor`								(without inlining)
mln	1315	2615	3915	5215	6515	7815	9115	10415	11715
mlnall	4959	9859	14759	19659	24559	29459	34359	39259	44159
spmax	63	63	63	63	63	63	63	63	63
	`foldl'xor`								(with inlining)
mln	1011	2011	3011	4011	5011	6011	7011	8011	9011
mlnall	4149	8249	12349	16449	20549	24649	28749	32849	36949
spmax	62	62	62	62	62	62	62	62	62
	`foldr` using `xor`								(without inlining)
mln	1115	2215	3315	4415	5515	6615	7715	8815	9915
mlnall	4056	8056	12056	16056	20056	24056	28056	32056	36056
spmax	66	66	66	66	66	66	66	66	66
	`foldrxor`								(with inlining)
mln	811	1611	2411	3211	4011	4811	5611	6411	7211
mlnall	3044	6044	9044	12044	15044	18044	21044	24044	27044
spmax	69	69	69	69	69	69	69	69	69

Since inlining copies into an abstraction (in addition into a recursive definition), our theoretical results do not give good guarantees on the space behavior and also do not guarantee that the transformation is space safe. However the LRPi is used to generate the table above and enables an analysis of space behavior of the inlined versions compared to the original ones.

For `foldl` the runtime decreases by at most 25 percent after inlining (this can be verified by estimating functions for both data rows and then using the limit on the division of both functions), since the number of reduction steps is decreased by a constant for each list element. In contrast the inlining increases the space consumption by a constant: Space consumption is linear in the length of the input list, which is caused by the left-associativity of `foldl` since we get a linear number of nested `case`-expressions caused by the `xor`. The constant increase of space consumption after inlining is caused by the constant additional space that is needed by the inlined `xor`-function.

In the case of `foldl'` the accumulator is evaluated each time and therefore no nested `case`-expressions are constructed. The space consumption is constant and a little bit better than the not-inlined variant. This is caused by the directly evaluated accumulator. The runtime improves by at most 23 percent.

For `foldr` the inlining improves the runtime by at most 28 percent. The space consumption also only increases by a constant (similar to the `foldl'`-variant). This is caused by the right-associativity of `foldr`: Since `xor` is strict in the first argument, `foldr` runs over the whole list, but depending on the left argument, `xor` either evaluates the second argument or returns the argument. Since the list is lazily generated and contains only `False`-elements (up to one occurrence), each element gets directly generated and consumed and therefore only constant space is needed.

The example suggests that it is a good idea to invest a limited amount of space for the runtime, since for `foldl'` and `foldr` the runtime improves by a good percentage while the space consumption only increases by an additive constant. This experiment shows a nice behavior in the considered empty context, but does not show the behavior in other contexts or other uses of the functions. Moreover the table above shows the difference between mln and mlnall, a lot of rules needed by the abstract machine M1 and also by the calculus LRP are only subsidiary (e.g. shifting of `letrec`-environments). Thus for further examples we omit the mlnall-values.

A similar example can be constructed using Peano numbers and addition instead of Boolean-values and `xor`. As combining function we use the lazy Peano-addition `padd` (see Section 2.6), as neutral element zero and as input list a list of zeroes, that again is generated using the take-function/list-generator approach. I.e. in Haskell-notation we compare `foldl 0 padd (take` k `[0,0..])` with `foldl'` and `foldr` with the same arguments, each with the inlined variant. The inlined variants `foldlpadd`, `foldl'padd` and `foldrpadd` are constructed analogous to Definition 3.20.

k	100	200	300	400	500	600	700	800	900
	`foldl` using `padd`								(without inlining)
mln	1107	2207	3307	4407	5507	6607	7707	8807	9907
spmax	258	458	658	858	1058	1258	1458	1658	1858
	`foldlpadd`								(with inlining)
mln	806	1606	2406	3206	4006	4806	5606	6406	7206
spmax	561	1061	1561	2061	2561	3061	3561	4061	4561
	`foldl'` using `padd`								(without inlining)
mln	1207	2407	3607	4807	6007	7207	8407	9607	10807
spmax	60	60	60	60	60	60	60	60	60
	`foldl'padd`								(with inlining)
mln	906	1806	2706	3606	4506	5406	6306	7206	8106
spmax	63	63	63	63	63	63	63	63	63
	`foldr` using `padd`								(without inlining)
mln	1107	2207	3307	4407	5507	6607	7707	8807	9907
spmax	58	58	58	58	58	58	58	58	58
	`foldrpadd`								(with inlining)
mln	806	1606	2406	3206	4006	4806	5606	6406	7206
spmax	65	65	65	65	65	65	65	65	65

In contrast to the example using `xor`, we see that inlining increases the space consumption for `foldl` by a multiplicative constant of at most 2.5, caused by the lazy Peano-addition that leads to unevaluated additions. For `foldl` the runtime improves by at most 28 percent. For `foldl'` the runtime improves by at most 25 percent and for `foldr` inlining improves by at most 28 percent, while the space consumption for both is only increased by an additive constant.

This example shows that the impact on the space behavior may depend on little details that need to be considered before an increase of space consumption is tolerated for an improvement of the runtime.

Variants of `reverse`

The naive implementation of `reverse` as given in Section 2.6, that reverses the order of the elements in a given list, is now compared with the variant `reverse'`. For `reverse'` the worker-wrapper approach is used, where `reversew` is the worker-function and `reverse'` the wrapper-function (compare (HH14) for more details on the improvement property of this approach).

Definition 3.21 (Variant of Reverse-Function for Lists `reverse'`)
We give the definition of `reverse'`, a variant of the function `reverse` for lists using the worker-function `reversew`:

$$\begin{aligned}
&\texttt{reverse'} = \lambda xs.\texttt{reversew}\ []\ xs\\
&\texttt{reversew} = \lambda xs, ys.\texttt{case}\ ys\ \texttt{of}\ \{([] \to xs)\\
&\qquad\qquad\qquad\qquad\qquad\qquad ((z:zs) \to \texttt{reversew}\ (z:xs)\ zs)\}
\end{aligned}$$

To create the input list `replicate` is used and using `last` it is ensured that the body of the whole list is evaluated.

k	50	100	150	200	250	300	350	400
	`last (reverse (replicate` k `True))`							
mln	4230	15955	35180	61905	96130	137855	187080	243805
spmax	462	862	1262	1662	2062	2462	2862	3262
	`last (reverse' (replicate` k `True))`							
mln	457	907	1357	1807	2257	2707	3157	3607
spmax	100	150	200	250	300	350	400	450

We see that `reverse` has quadratic runtime that is caused by the left-associativity of `(++)` and using the worker-wrapper approach `reverse'` avoids this using the extra argument of `reversew` and therefore has linear runtime.

Since `reverse` only goes through the intermediate lists, asymptotical no additional space compared to `reverse'` is needed, though `reverse'` improves the space consumption by at most 87.5 percent.

Fusion of `concat` and `map` with Different Kinds of Garbage Collection

A common practice in functional languages is the composition of functions, since the readability and also the maintainability of programs is often improved. Especially the explicit recursions are hidden and the program can be often read very abstract without seeing too much unneeded details.

However the composition of functions lead to intermediate structures and therefore the runtime and space consumption might be increased, if a practical garbage collector (i.e. a non-eager garbage collection) is used. The Glasgow Haskell Compiler (GHC) uses the so called short cut fusion as introduced in (GLJ93). This approach eliminates such intermediate tree and list structures to gain a better runtime and to reduce the needed space.

As shown in (JV05), short cut fusion might be unsafe if `seq` is used, but in many cases this approach works and is used by the GHC. Moreover (Sve02) shows that this approach might increase sharing and therefore a part of the memory is longer used. Thus it may increase the space consumption.

For a function composition we use `comp` as defined in Section 2.6, where also the definition of `concat` is given. The straightforward fusion of `concat` and `map` leads to the following definition of `concatMap`:

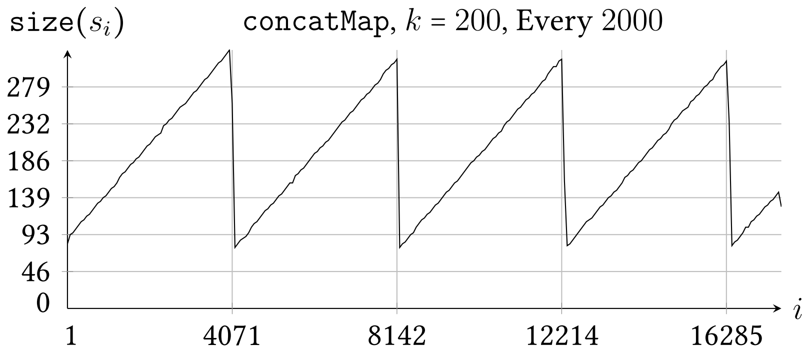
Definition 3.22 (Fusion of `concat` and `map`)
The fusion of `concat` and `map` is defined as follows:

$$\begin{aligned}\texttt{concatMap} = \lambda f, xs.(&\texttt{foldr}\\ &(\lambda x, y.\texttt{foldr}\ (\lambda z, zs.(z : zs))\ y\ (f\ x))\\ &[]\ xs)\end{aligned}$$

We now compare `(comp concat map) tail` with `concatMap tail`. As input a list containing k inner lists of the form `[True,True]` is used, generated by a list-generator/take-function approach, where the LRPi is configured to count each Peano-number with `size` 1.

k	100	200	300	400	500	600	700	800	900
	Unfused								
mln	2815	5615	8415	11215	14015	16815	19615	22415	25215
spmax	97	97	97	97	97	97	97	97	97
	Fused								
mln	2609	5209	7809	10409	13009	15609	18209	20809	23409
spmax	82	82	82	82	82	82	82	82	82

The fusion improves the runtime only insignificant and also the space consumption in both cases is only constant and therefore only improved by a constant. However we want to see if the frequency of the garbage collector has an impact, where the following diagram shows the space consumption if the garbage collector is only active every 2000th reduction step.



The rarer the garbage collector runs the higher is the space consumption. If we turn off the garbage collector, then we have the following space consumption for the unfused and fused variant:

k	100	200	300	400	500	600	700	800	900
	Unfused without Garbage Collection								
spmax	727	1327	1927	2527	3127	3727	4327	4927	5527
	Fused without Garbage Collection								
spmax	595	1095	1595	2095	2595	3095	3595	4095	4595

Thus, fusion improves the space consumption by at most 16 percent if garbage collection is turned off. With regard to LRPgc the advantage concerning space consumption of the fused versions over the unfused versions of the above examples is only an additive constant, but there is a higher advantage if garbage collection is turned off. This is interesting for the practice, because a practical garbage collector has longer cycles than the eager garbage collection of LRPgc.

Impact of Sharing on Space Consumption

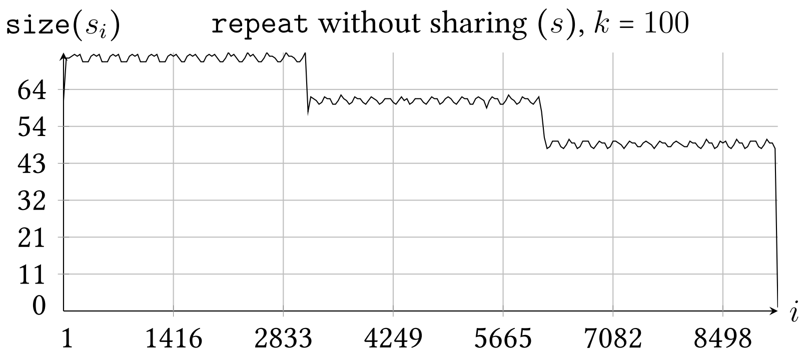
To share, or not to share, that is the question. While improving the runtime, sharing might lead to an increase of space consumption, since intermediate results might be stored longer in memory than without sharing.

In the following example we analyze the impact of sharing on runtime and space consumption. In the following `allTrue` is used, a direct implementation of `all (==True)` (in Haskell-notation), that leads to less β-reductions and therefore too less distortion of the results. For more details see Section 2.6.

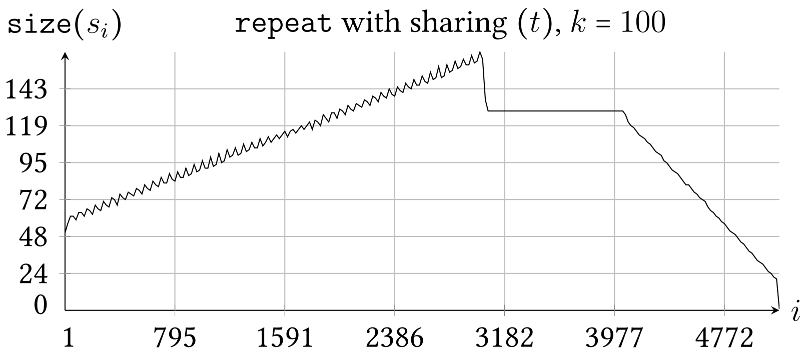
The LRP-program s is given in Haskell-notation where k is a given Peano number. The program generates three equal lists that each contain k `True`-elements and applies `allTrue` to those lists, yielding `True` for each list and then computes the logical and of all of these `True`-values, yielding `True`. Intuitively, the three lists a merged into one and then and is folded over this list from the left to the right. The LRP-program t is similar to s but (cse) was applied, thus t shares the arguments calling the function f.

```
s := letrec f = λy.take k (repeat y)
       in and (allTrue (f True)) (and (allTrue (f True))
                                      (allTrue (f True)))
t := letrec f = λy.take k (repeat y), x = (f True)
       in and (allTrue x) (and (allTrue x) (allTrue x))
```

Without sharing each of the three lists containing k `True`-elements is evaluated independently one after each other. Since `allTrue` only traverses those lists a single time, each list element can be directly garbage collected after creation, leading to the following diagram. Since from list to list more and more overhead can be removed, the number of down-steps in the curve directly corresponds to the number of lists containing k `True`-elements.



Since `allTrue` is applied on the lists containing k `True`-elements, the binding x in the LRP-expression t contains an evaluated list containing k `True`-elements. Caused by the sharing here the garbage collector cannot remove the list containing k `True`-elements, hence the list is evaluated during the whole runtime and therefore increases the space consumption. As expected the sharing decreases the runtime:



The differences in space consumption and runtime are visible if we compare different list lengths of the lists containing k `True`-elements. As expected the space consumption directly depends on k if sharing is used and is constant without sharing. The runtime improves by at most 42 percent using sharing, where the reason for this constant and not asymptotic improvement is, that the same number of `True`-elements are traversed for both the shared and the unshared variant.

k	100	200	300	400	500	600	700	800	900
	`repeat` without sharing (s)								
mln	2424	4824	7224	9624	12024	14424	16824	19224	21624
spmax	75	75	75	75	75	75	75	75	75
	`repeat` with sharing (t)								
mln	1416	2816	4216	5616	7016	8416	9816	11216	12616
spmax	167	267	367	467	567	667	767	867	967

The above example can be generalized, i.e. l lists containing k `True`-elements that are combined using $l - 1$ occurrences of `and`. Then an intuitive approach is a helper function `h` that generates the described structure, taking `f True` or `f x` as parameter. But then the β-reduction destroys the sharing and to solve this, the binding x in the sharing of the example above needs to be referenced directly by `h`. However this is not a common way of programming and that the generalization of the example has a better space consumption than the special case above, shows that minor details might have a great impact on the space consumption.

While the above example used Boolean-values, in the following we give an example for common subexpression elimination using append (++). Let xxs be a list. We compare

$$(xxs \mathbin{++} xxs) \mathbin{++} (xxs \mathbin{++} xxs)$$

with its shared variant

$$\texttt{let}\ xxs' = xxs\ \texttt{in}\ (xxs' \mathbin{++} xxs') \mathbin{++} (xxs' \mathbin{++} xxs')$$

written here in Haskell-notation. To force the evaluation an outer `last` is used.

The first expression has four separate occurrences of the list xxs, whereas the second expression shares xxs, where different lengths are used for xxs in the experiments. We assume that xxs only contains Boolean-values, i.e. no Peano numbers and therefore again configure the LRPi so that Peano numbers are not counted by the space measure.

k	100	200	300	400	500	600	700	800
	Append without sharing							
mln	3621	7221	10821	14421	18021	21621	25221	28821
spmax	66	66	66	66	66	66	66	66
	Append with sharing							
mln	2409	4809	7209	9609	12009	14409	16809	19209
spmax	253	453	653	853	1053	1253	1453	1653

As expected the space consumption is increased linearly by the sharing, since the list xxs with length k is evaluated by the first append and remains in memory until the computation finishes. However the sharing improves the runtime by at most 33.3 percent, because the list xxs does not need to be calculated at each time. The impact of sharing w.r.t. runtime depends on the definition of xxs, here we used `replicate` to create a list of `True`-values.

If Peano numbers are counted by the space measure, then the used take-function/list-generator approach leads to a distortion: For both shared and unshared the space consumption is linear caused by k implemented as Peano number. Moreover this distortion leads to the effect that for $k \leq 12$ the shared variant has a lower space consumption than the unshared version.

k	1	2	12	13	14	200	400	600	800
	Append without sharing								
mln	57	93	453	489	525	7221	14421	21621	28821
spmax	67	68	78	79	80	266	466	666	866
	Append with sharing								
mln	33	57	297	321	345	4809	9609	14409	19209
spmax	56	57	77	79	81	453	853	1253	1653

However both with Peano numbers counted by the space measure or not, the results are consistent with the claim that common subexpression elimination (cse) is a time improvement (proved in (SS15)) and show that (cse) and an increase of sharing in general may increase the (maximal) space usage.

So the question, if it is a good idea to share or not to share, depends on the scenario. If it is clear, that the space increase will be not that high, than the sharing is a good idea to improve the runtime. However the examples showed that sharing is a space-critical time improvement, since the space consumption may increase even asymptotically.

Combination of Optimizations

In this section we consider a program that combines the different optimizations of the sections before.

We use a simple program that uses all those optimizations in a realistic and practical scenario. Using straightforward datatype-definitions for tuples containing 2 and 4 elements, we assume that the input of the program is a list of 4-tuples describing an article of a small shop: $($*article number*, *price without tax*, *tax*, *in stock*$)$

Now we assume that the shop has multiple lists of such 4-tuples all combined into one list xxs, that is the input for the program. The goal is to compute a pair containing each a pair of article number and price including taxes for the article with the lowest and highest price including taxes, i.e. $((artno_1, p_1), (artno_2, p_2))$ where p_1 is the lowest price including taxes of xxs and p_2 the highest.

The unoptimized program is defined as follows, where some Haskell-notations are used for better readability and needed functions are defined in Section 2.6:

```
letrec min = fBySnd plower ((comp (map (λ(n,p,t,s).(n,padd p t)))
                                  (filter (λ(n,p,t,s).s)))
                                          (foldl (++) [] xxs)),
       max = fBySnd pgreater ((comp (map (λ(n,p,t,s).(n,padd p t)))
                                    (filter (λ(n,p,t,s).s)))
                                            (foldl (++) [] xxs))
   in seq min (seq max (min,max))
```

`foldl` is used to flatten xxs, yielding a single list containing all articles. Then `filter` is applied to eliminate all articles that are not in stock and `map` is applied afterwards to transform the 4-tuples to pairs, that only contain the article numbers together with the prices including taxes. Note that `filter` and `map` are implemented using function-composition, making it easier to apply fusion as optimization later. Finally the result is calculated using `fBySnd`. For all numbers Peano numbers are used – eventually the LRPi is configured to measure each Peano number with size 1 for a realistic space measurement. This is also the reason for the two `seq`-expressions. Also note that it is a good idea to give the program an input with articles that all have the same prices, because then the calculation of minimal and maximal prices are forcing a strictness and allow an easier analysis.

At first glance we see that there is a big opportunity for improvement using common subexpression elimination, since the most parts of the calculation of min and max are identical. Also `map` and `filter` can be fused using the following definition:

Definition 3.23 (Fusion of `map` and `filter`)
The fusion of `map` and `filter` is defined as follows:

$$
\begin{aligned}
\texttt{mapFilter} = \lambda f, p, xxs.\texttt{case}\ xxs\ \texttt{of}\ \{ & \\
& ([] \to []) \\
& ((x : xs) \to \texttt{case}\ (p\ x)\ \texttt{of}\ \{ \\
& \qquad (\texttt{True} \to ((f\ x) : (\texttt{mapFilter}\ f\ p\ xs))) \\
& \qquad (\texttt{False} \to \texttt{mapFilter}\ f\ p\ xs)\})\}
\end{aligned}
$$

Moreover `foldl` is not a good idea in combination with `(++)` because of its left-associativity, hence an optimization is the replacement of `foldl` by `foldr`.

Let k be the length of xxs, then these are the results for the different combinations of the described optimizations calculated by the LRPi:

k	100	200	300	400	500	600	700	800
	Unoptimized							
mln	3439	8699	15759	24619	35279	47739	61999	78059
spmax	299	369	439	509	579	649	719	789
w/o GC	718	1758	3398	5638	8478	11918	15958	20598
	Optimizations: `foldr` instead of `foldl`							
mln	2809	5639	8469	11299	14129	16959	19789	22619
spmax	268	278	288	298	308	318	328	338
w/o GC	518	748	978	1208	1438	1668	1898	2128
	Optimizations: Fusion							
mln	3369	8569	15569	24369	34969	47369	61569	77569
spmax	281	351	421	491	561	631	701	771
w/o GC	617	1617	3217	5417	8217	11617	15617	20217
	Optimizations: (cse)							
mln	2491	5916	10241	15466	21591	28616	36541	45366
spmax	256	316	376	436	496	556	616	676
w/o GC	473	1018	1863	3008	4453	6198	8243	10588
	Optimizations: `foldr` instead of `foldl`, Fusion							
mln	2739	5509	8279	11049	13819	16589	19359	22129
spmax	250	260	270	280	290	300	310	320
w/o GC	415	605	795	985	1175	1365	1555	1745
	Optimizations: `foldr` instead of `foldl`, (cse)							
mln	2176	4386	6596	8806	11016	13226	15436	17646
spmax	275	315	355	395	435	475	515	555
w/o GC	381	508	648	788	928	1068	1208	1348
	Optimizations: `foldr` instead of `foldl`, Fusion, (cse)							
mln	2141	4321	6501	8681	10861	13041	15221	17401
spmax	264	304	344	384	424	464	504	544
w/o GC	322	433	553	673	793	913	1033	1153

The quadratic runtime of the unoptimized variant is caused by `foldl` in combination with `(++)` and using the limit we determine that the fold-optimization improves the space consumption by at most 85 percent – note that the left-associativity of `foldl` does not asymptotically affect the space consumption for this program, since `(++)` only runs through the lists.

The results for all cases were we compare a variant with fusion with a variant without fusion, are compatible to the results in Section 3.6.2: Using `mapFilter` instead of composed `map` and `filter` improves the runtime by an additive constant. Using eager garbage collection, the space consumption also is improved by an additive constant and by at most 17 percent if the garbage collector is turned off.

Common subexpression elimination also for this program leads to a list that is stored in memory during the whole calculation, while without common subexpression elimination, the list elements can be directly garbage collected after they are created. However the runtime and space problems introduced by `foldl` in combination with `(++)` are so strong, that only using (cse) as optimization improves the program w.r.t. runtime and space consumption in any case. The reason is the duplicated calculation of the same list using `foldl` with `(++)` in the unoptimized variant, while the (cse)-variant only calculates this list once. As expected this improvement of runtime and space consumption is not of asymptotic relevance, still the (cse)-optimized variant needs quadratic runtime and the space consumption is improved by at most 14 percent. However the runtime improving but space worsening transformation (cse) is also a space improvement if `foldl` is used here.

In the case the fold-optimization is applied, we see that (cse) improves the runtime by at most 22 percent, while the space consumption is increased by a multiplicative constant of at most 4.

In summary, the fold-optimization and fusion should be applied in any case, since the runtime is improved from quadratic to linear and the space consumption is improved by at most 85.7 percent. The addition of common subexpression elimination increases the space consumption by a multiplicative constant of at most 4 and therefore can be applied, if the runtime is critical and should not be applied if the available memory is critical.

4
Total Garbage Collection in LR

In the previous chapter space optimizations in LRP using an eager garbage collector were considered. Also the impact of garbage collection was considered using an interpreter, where the garbage collector was only active every k-th reduction step for a specified k or completely turned off. In this chapter we go in the other direction: A garbage collector is used, that is able to remove all unneeded subexpressions in one step in contrast to a implementable garbage collector working only on top-`letrec` (i.e. heap-bindings).

Of course such a garbage collector is not implementable, but it provides a foundation for space analyses that are completely independent of the used garbage collector. All the proofs in the previous sections need to be reconsidered if the garbage collector is exchanged. Moreover such a garbage collector allows to analyze a distortion-free scenario and the showed space improvements are near at the optimal scenario and therefore more independent. Especially with respect to the improvement of garbage collectors these results are helpful orientations.

Such an garbage collector is not correct, since it might destroy the semantics if applied only to subexpressions. However it is usable to provide theoretical results.

In this chapter we first define a so-called total garbage collector and an appropriate space measure for an evaluation sequence. We prove a context lemma to reduce the amount of contexts in the proofs of space improvements and then analyze the space properties of several transformations. As calculus LR is used instead of LRP, since this eases the proofs and moreover the results are more general, since we have a wider scope of programs and this fits to the generalization-approach of a total garbage collector compared to a deterministic eager garbage collector. This chapter is based on (SSD19b).

4.1 Definition of Space Improvement and Equivalence

The idea of total garbage collection is to remove all subexpressions that do not contribute to the termination of the expression. If a subexpression does contribute to the termination of an expression is clearly undecidable.

Practical garbage collectors may have different abilities to predict the non-usage of subexpressions. If we use a total garbage collection, that removes any unneeded subexpressions, then this garbage collection is independent of the various implementations of garbage collections during evaluations and thus of independent value. Our view is that all practical garbage collections are approximations of different strength of the total garbage collection.

Garbage collectable positions are defined as positions, where the corresponding subexpression can be replaced by a non-terminating constant Bot without changing the overall convergence. We now use garbage collectable position to recognize subexpressions that are garbage w.r.t. total garbage collection. Positions in terms are defined as usual as lists p of positive integers representing the path from the root to the tree-positions. $t[p \mapsto t']$ means the term that is constructed from t by replacing the position p with t'. We use Bot instead of $\bot$ for formal reasons later.

Definition 4.1 (Constant Bot and Garbage Collectable Position)
Let Bot be a constant that does not converge and has all types. A position p of an expression t is called *garbage-collectable* iff $t{\downarrow} \iff t[p \mapsto Bot]{\downarrow}$.

The sequence of replacing garbage-collectable positions is irrelevant:

Lemma 4.1 (Irrelevance of Order of Replacing Garbage-Collectable Positions)
Let $p_1, \dots, p_n$ be some garbage collectable positions of t. Then the following holds:

$$t{\downarrow} \iff t[p_1 \mapsto Bot, \dots, p_n \mapsto Bot]{\downarrow}$$

Proof
Let $t' := t[p_1 \mapsto Bot, \dots, p_n \mapsto Bot]$. We have $t' \leq_c t$, hence $t'{\downarrow}$ implies $t{\downarrow}$.

Now assume $t{\downarrow}$ and $t'{\uparrow}$. Then the normal-order reduction of t' must put some successor-position of p_k in the expressions of the reduction sequence in a reduction context, independent of the other positions. Then p_k cannot be a garbage-collectable position, which is a contradiction. Hence $t'{\downarrow}$.

Now we can define a relation for total garbage collection:

Definition 4.2 (Total Garbage Collection (tgc))
Let s be an LR-expression. Then $s \xrightarrow{tgc} s'$ is defined by total garbage collection, i.e. $s' = s[p_1 \mapsto Bot, \dots, p_n \mapsto Bot]$ where the p_i, $i = 1, \dots, n$ are all minimal garbage collectable positions of s. If s is not changed, then it is in *tgc-normalform*.

Note that (tgc) is in general not correct. An example is the expression (`Cons True Nil`), where (`Cons True Nil`) $\xrightarrow{tgc}$ (`Cons` Bot Bot). Consider the following context C: `case` $[\cdot]$ `of` $\{$(`Nil` $\to$ `Nil`) ((`Cons` x y) $\to$ x)$\}$. Then we have $C[$(`Cons True Nil`)$]{\downarrow}$ and $C[$(`Cons` Bot Bot)$]{\uparrow}$, hence (tgc) is not a contextual equivalence.

We reuse Definition 3.5 and extend it so that Bot is not counted, since this represents a subexpression that was removed by an application of (tgc) before:

Definition 4.3 (LR Size of Expressions `size`)
The size `size`(s) of an LR-expression s is defined as in Definition 3.5, where it is exploited that the syntax of LR is a subset of the syntax of LRP.
For Bot we define `size`$(Bot) = 0$.

The measure `sizetgc` also does not count the size of garbage collectable positions:

Definition 4.4 (LR Size of Non-Garbage-Collectable Expressions `sizetgc`)
Let s be an expression. Then $\texttt{sizetgc}(s) := \texttt{size}(s')$ where $s \xrightarrow{tgc} s'$.

Based on the measure `sizetgc` for expressions, the space measurement for evaluation sequences is defined as follows:

Definition 4.5 (LR Space Measure *sps*)
The space measure $\mathit{sps}(s)$ for the normal-order reduction sequence of a closed LR-expression s is defined as follows:

$$\mathit{sps}(s) = \max\{\texttt{sizetgc}(s_i) \mid \text{ where } s_i \text{ are the expressions in a normal-order reduction sequence of } s\}$$

Now we can give a definition of total space improvements and equivalences.

Definition 4.6 (LR Total Space Improvement and Equivalence)
Let s and t be two LR-expressions with $s \sim_c t$.

– s is a *total space improvement* of t, $s \leq_{\mathit{sps}} t$, if for all contexts $\mathbb{C}$ we have: $\mathit{sps}(\mathbb{C}[s]) \leq \mathit{sps}(\mathbb{C}[t])$
– s is *totally-space-equivalent* to t, $s \sim_{\mathit{sps}} t$, if for all contexts $\mathbb{C}$ we have: $\mathit{sps}(\mathbb{C}[s]) = \mathit{sps}(\mathbb{C}[t])$

A transformation $\xrightarrow{trans}$ is called a *total space improvement* (*total space equivalence*) if $t \xrightarrow{trans} s$ implies that s is a total space improvement of (totally-space-equivalent to, respectively) t. We often say s *space-improves* t.
We write $s \leq_{X,\mathit{sps}} t$ ($s \sim_{X,\mathit{sps}} t$) for a context class X, to denote that the definition is as above but restricted to context class X. E.g. for $s \leq_{\mathbb{R},\mathit{sps}} t$, we require that the inequality $\mathit{sps}(\mathbb{R}[s]) \leq \mathit{sps}(\mathbb{R}[t])$ holds for all reduction contexts $\mathbb{R}$.
Note that $t \geq_{\mathit{sps}} s$ is sometimes used instead of $s \leq_{\mathit{sps}} t$.

This definition of total space improvements applies for libraries, since the functions in a library are intended for a general purpose and stored in a precompiled form in memory to be used by multiple independent programs. Hence this notion is especially useful for optimizations where the specific use of a function is not known.

However a compiler can perform further optimizations, if the specific use of a function is known. This motivates the following definition of a weaker variant that only applies in the immediate evaluation situations, where the whole program is known.

Definition 4.7 (Opportunistic Total Space Improvement)
Let s, t be two LR-expressions with $s \sim_c t$. If $\mathit{sps}(s) \leq \mathit{sps}(t)$ holds, then s *opportunistically totally space improves* t. We also say that s is an *opportunistic total space improvement* of t.

In contrast to the standard notion of space improvement as in Definition 3.7 or in (GS01, GS99, SSD18) the total space improvement property $s \leq_{\mathit{sps}} t$ does not imply $\texttt{size}(s) \leq \texttt{size}(t)$ and also it does not imply $\mathit{FV}(s) \subseteq \mathit{FV}(t)$. This is an advantage, since it permits a more general definition and view of space consumption and space improvements. Also this notion of total space improvement permits more interesting transformations. However this notion implies $\texttt{sizetgc}(s) \leq \texttt{sizetgc}(t)$:

Lemma 4.2 (Total-Space Improvement-Property Implies Less or Equal `sizetgc`)
If s is a total space improvement of t, i.e. $s \leq_{sps} t$, then $\texttt{sizetgc}(s) \leq \texttt{sizetgc}(t)$.

Proof
We show that if $\texttt{sizetgc}(s) > \texttt{sizetgc}(t)$ and $s \sim_c t$, then in general s cannot be a total space improvement of t.

Let us assume that s is a total space improvement of t and let $m = \mathit{sps}(t) - \mathit{sps}(s) \geq 0$. Also let $s_0 := (\texttt{seq}^*\ s \ldots s)$ and $t_0 := (\texttt{seq}^*\ t \ldots t)$, where $\texttt{seq}^*$ is an iterated `seq` and the number of occurrences of $m + 2$.

Then $\mathit{sps}(s_0) = \mathit{sps}(s) + (m+1) \cdot (1 + \texttt{size}(s))$, since the normal-order reduction first only modifies the leftmost s, then normal-order-reduces $(\texttt{seq}^*\ s \ldots s)$ with one s-occurrence removed. The same holds for t_0: $\mathit{sps}(t_0) = \mathit{sps}(t) + (m+1) \cdot (1 + \texttt{size}(t))$

Since $\texttt{size}(s) > \texttt{size}(t)$ we obtain:

$$\mathit{sps}(s_0) - \mathit{sps}(t_0) = -m + (m+1) \cdot (\texttt{size}(s) - \texttt{size}(t)) > 0$$

This is a contradiction to the assumption that s is a total space improvement of t.

Note that (tgc) is in general not a total space improvement, since it is not correct as showed above. We now analyze different forms of garbage collection.

Definition 4.8 (Garbage Collecting Transformation (gcg))
Let (gcg) be a (garbage collecting) transformation that is an approximation of (tgc), i.e. (gcg) : $s \to s'$ holds for LR-expressions s and s', if and only if:

1. There is a set of positions $p_1, \ldots, p_n$, where every position p_i is the same or below some garbage collectable position.
2. $s' = s[p_1 \mapsto Bot, \ldots, p_n \mapsto Bot]$

Using (gcg) we show that deterministic (gc) as used in Definition 3.2 and also more general forms of garbage collection are total space improvements.

Theorem 4.1 (Space Improvement Property of (gcg))
In any case (gcg) is an opportunistic total space improvement. If (gcg) is correct, then (gcg) is a total space improvement.

Proof
We have to show the total space improving property. Let s be an expression and let $\mathbb{C}$ be a context and $s \xrightarrow{gcg} s'$.

Since the (gcg)-positions are removed by (tgc), the general reduction diagram is as follows:

$$\begin{tikzcd}
\mathbb{C}[s] \arrow[r, "gcg"] \arrow[d, "tgc"'] & \mathbb{C}[s'] \arrow[dl, dashed, "tgc"] \\
s_1 &
\end{tikzcd}$$

We see that s_1 is a common expression in the reduction sequence and we have $\texttt{sizetgc}(\mathbb{C}[s]) \geq \texttt{sizetgc}(\mathbb{C}[s'])$. Hence $\mathit{sps}(\mathbb{C}[s]) \geq \mathit{sps}(\mathbb{C}[s'])$.

4.2 Context Lemma for Total Space Improvement

The total space improvement definition considers all contexts. Therefore we use the same approach as in Chapter 3 using a context lemma, that reduces the amount of cases that need to be considered in the case analyses of interferences between normal order reductions and transformations. Since we want to use total garbage collection, we use the measure `sizetgc` here.

First of all, we analyze the `sizetgc` of WHNFs:

Lemma 4.3 (`sizetgc` of WHNFs)
For every WHNF s, we have $\texttt{sizetgc}(s) = 1$.

Proof
A WHNF s is in one of the following forms:

1. $(\lambda x.t)$ or $(c\, t_1\, \dots\, t_n)$ (below called simple WHNF)
2. $\texttt{letrec}\ E\ \texttt{in}\ t$, where t is a simple WHNF
3. $\texttt{letrec}\ E\ \texttt{in}\ x$, where x is bound in E to a simple WHNF

After total garbage collection, the expressions are of the following forms:

1. $(\lambda x.Bot)$ or $(c\, Bot\, \dots\, Bot)$ (below called simple garbage collected WHNF)
2. $\texttt{letrec}\ E\ \texttt{in}\ t'$, where t' is a simple garbage collected WHNF
3. $\texttt{letrec}\ E\ \texttt{in}\ x$, where x is bound in E to a simple garbage collected WHNF

Hence $\texttt{sizetgc}(s)$ is 1.

For the context lemma we also need the following notation:

Definition 4.9 (Notations for `sizetgc` w.r.t. Contexts)
Let s, t be LR-expressions. If we have $\texttt{sizetgc}(\mathbb{C}[s]) \leq \texttt{sizetgc}(\mathbb{C}[t])$ for all contexts $\mathbb{C}$, then we denote this as $s \leq_{\mathbb{C},\texttt{sizetgc}} t$ (we sometimes also write $t \geq_{\mathbb{C},\texttt{sizetgc}} s$). If for all contexts $\mathbb{C}$, we have $\texttt{sizetgc}(\mathbb{C}[s]) = \texttt{sizetgc}(\mathbb{C}[t])$, then we denote this as $s \sim_{\mathbb{C},\texttt{sizetgc}} t$.

The following lemma is needed to show the correctness of the context lemma:

Lemma 4.4 (Portability of `sizetgc` to Multicontexts)
If M is a multicontext with n holes and s_i, t_i are LR-expressions with $s_i \leq_{\mathbb{C},\texttt{sizetgc}} t_i$ for all i, then also $M[s_1, \dots, s_n] \leq_{\mathbb{C},\texttt{sizetgc}} M[t_1, \dots, t_n]$.

Proof
We show the claim by induction on the number n of holes.

$M[s_1, \dots, s_n] \leq_{\mathbb{C},\texttt{sizetgc}} M[t_1, \dots, t_n]$ follows from:

$$M[s_1, \dots, s_{n-1}, s_n] \leq_{\mathbb{C},\texttt{sizetgc}} M[s_1, \dots, s_{n-1}, t_n] \leq_{\mathbb{C},\texttt{sizetgc}} M[t_1, \dots, t_{n-1}, t_n]$$

The first holds by assumption on s_n, t_n and the second by the induction hypothesis.

The context lemma is similar to Lemma 3.6 but caused by the total garbage collection more general:

Lemma 4.5 (Context Lemma for Total Space Improvements)
If $s \sim_c t$, $s \leq_{\mathbb{R},sps} t$ and $s \leq_{\mathbb{C},\mathtt{sizetgc}} t$, then $s \leq_{sps} t$.

Proof
Let M be a multicontext. We prove the more general claim that if $M[s_1, \ldots, s_n]$ and $M[t_1, \ldots, t_n]$ are closed, $M[s_1, \ldots, s_n]{\downarrow}$ and $s_i \leq_{R,sps} t_i$ holds for all i, then $M[s_1, \ldots, s_n] \leq_{sps} M[t_1, \ldots, t_n]$.

By the assumption that $s_i \sim_c t_i$, we have $M[s_1, \ldots, s_n] \sim_c M[t_1, \ldots, t_n]$ and thus $M[s_1, \ldots, s_n]{\downarrow} \iff M[t_1, \ldots, t_n]{\downarrow}$. The induction proof is on the number of LR-reduction steps of $M[t_1, \ldots, t_n]$ and as a second parameter on the number of holes of M. We distinguish the following cases:

1. If no hole of M is in a reduction context, then there are two cases:
   - $M[t_1, \ldots, t_n]$ is a WHNF. The context M itself must be a WHNF, since otherwise there is a hole of M in a reduction context. Then $M[s_1, \ldots, s_n]$ is also a WHNF and by the assumption, the following inequation holds:
   $$1 = sps(M[s_1, \ldots, s_n]) \leq sps(M[t_1, \ldots, t_n])$$
   - The reduction step is $M[t_1, \ldots, t_n] \xrightarrow{LR,a} M'[t'_1, \ldots, t'_{n'}]$ and we also have $M[s_1, \ldots, s_n] \xrightarrow{LR,a} M'[s'_1, \ldots, s'_{n'}]$ and the pairs (s'_i, t'_i) are renamed versions of pairs (s_j, t_j). This shows $sps(M'[s'_1, \ldots, s'_{n'}]) \leq sps(M'[t'_1, \ldots, t'_{n'}])$ by induction.

     The inequation $\mathtt{sizetgc}(M[s_1, \ldots, s_n]) \leq \mathtt{sizetgc}(M[t_1, \ldots, t_n])$ holds by Lemma 4.4 and the preconditions of this lemma, hence by computing the maximum, we obtain $sps(M[s_1, \ldots, s_n]) \leq sps(M[t_1, \ldots, t_n])$.
2. Some t_j in $M[t_1, \ldots, t_n]$ is in a reduction position. Then there is one hole, say i, of M that is in a reduction position w.r.t. only M. Let $M' = M[\cdot, \ldots, \cdot, t_i, \cdot, \ldots, \cdot]$. We then can apply the induction hypothesis, since the number of holes of M' is strictly smaller than the number of holes of M and the number of normal-order reduction steps of $M[t_1, \ldots, t_n]$ is the same as of $M'[t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n]$. We now obtain:
   $$sps(M[s_1, \ldots, s_{i-1}, t_i, s_{i+1}, \ldots, s_n]) \leq sps(M[t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n])$$
   Also since $M[s_1, \ldots, s_{i-1}, \cdot, s_{i+1}, \ldots, s_n]$ is a reduction context, the assumption yields:
   $$sps(M[s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n]) \leq sps(M[s_1, \ldots, s_{i-1}, t_i, s_{i+1}, \ldots, s_n])$$
   Hence $sps(M[s_1, \ldots, s_n]) \leq sps(M[t_1, \ldots, t_n])$.

For cases that do not change maximal space consumption, we adapt Lemma 4.5 as follows:

Lemma 4.6 (Context Lemma for Total Space Equivalences)
If $s \sim_c t$, $s \sim_{\mathbb{R},sps} t$ and $s \sim_{\mathbb{C},\texttt{sizetgc}} t$, then $s \sim_{sps} t$.

Proof
Follows by applying Lemma 4.5 in both directions.

The context lemmas also obviously hold for stronger context classes:

Proposition 4.1 (Applicability of Context Lemmas for Stronger Context Classes)
The context lemmas Lemma 4.5 and Lemma 4.6 also hold for all context classes that contain reduction contexts.
Especially both lemmas also hold for top and surface contexts.

In the following we show criterions for the context lemma requirements. First we show a property that is used below several times in variants.

Lemma 4.7 (Portability of Garbage Collectable Positions)
Let M_1 and M_2 be multicontexts, so that $M_1[s_1, \dots, s_n] \sim_c M_2[s_1, \dots, s_n]$ for all expressions $s_1, \dots, s_n$. If $p = p_{1,1}p_2$ is a garbage collectable position of $M_1[s_1, \dots, s_n]$ that points into s_i, where $p_{1,1}$ is the position of the i-th hole of M_1 and p_2 is the position in s_i, then for the position $p_{2,1}$ of i-th hole of M_2, also $p_{2,1}p_2$ is a garbage collectable position in $M_2[s_1, \dots, s_n]$.

Proof
The simple argument is:

$$\begin{array}{rcl} M_1[s_1, \dots, s_n] & \sim_c & M_1[s_1, \dots, s_i[p_2 \mapsto Bot], \dots, s_n] \\ & \sim_c & M_2[p_1, \dots, p_i[p_2 \mapsto Bot], \dots, s_n] \end{array}$$

Hence $M_2[s_1, \dots, s_i[p_2 \mapsto Bot], \dots, s_n] \sim_c M_2[s_1, \dots, s_n]$. The same for the direction from M_2 to M_1. Hence the claim holds.

Now we show a criterion for $\leq_{\mathbb{C},\texttt{sizetgc}}$, the first requirement of the context lemma:

Lemma 4.8 (Criterion for $\leq_{\mathbb{C},\texttt{sizetgc}}$)
Let s, t be LR-expressions, so that $s \sim_c t$, $s = \mathbb{C}_s[s_1, \dots, s_n]$ and $t = \mathbb{C}_t[s_1, \dots, s_n]$ and $\texttt{size}(\mathbb{C}_s) \leq \texttt{size}(\mathbb{C}_t)$. Moreover we assume that the translation T, $\mathbb{C}_s[r_1, \dots, r_k]$ to $\mathbb{C}_t[r_1, \dots, r_k]$, is correct for all r_j. We also assume that all positions in $\mathbb{C}_s, \mathbb{C}_t$ that are not the hole positions are reduction positions in the respective contexts.
Then $s \leq_{\mathbb{C},\texttt{sizetgc}} t$.

Proof
Since $s \sim_c t$, which implies $\mathbb{C}[s] \sim_c \mathbb{C}[t]$, the garbage-collectable positions in $\mathbb{C}$ are the same on the left and right hand side. Let p be a garbage-collectable position in $\mathbb{C}[\mathbb{C}_s[s_1, \dots, s_n]]$ that is in s and goes down to s_1 w.l.o.g. Since the translation T is correct, the position p can be split into $p_1p_{2,s}p_3$ where $p_{2,s}$ is the position of the first hole of $\mathbb{C}_s$. Using Lemma 4.7 we see that also $p_1p_{2,t}p_3$ is a garbage collectable

position where $p_{2,t}$ is the position of the hole of $\mathbb{C}_t$ and vice versa. For the position q_1 of s and t itself Lemma 4.7 also shows that q_1 in $\mathbb{C}[s]$ is garbage collectable if q_1 in $\mathbb{C}[t]$ is garbage collectable.

Hence there is a one-to-one-correspondence between the garbage collectable positions of s in $\mathbb{C}[s]$ and t in $\mathbb{C}[t]$. As a summary, the garbage collectable positions in $\mathbb{C}[s]$ and $\mathbb{C}[t]$ are in one-to-one-correspondence and either point to equal expressions or the expression on the s-side is not greater in size than the one of $\mathbb{C}[t]$. Thus $\texttt{sizetgc}(\mathbb{C}[s]) \leq \texttt{sizetgc}(\mathbb{C}[t])$.

An example for the situation in Lemma 4.8 is the beta-reduction as transformation, i.e. $((\lambda x.s)\ t) \to \texttt{letrec}\ x = t\ \texttt{in}\ s$ and the two contexts are $((\lambda x.[\cdot]_1)\ [\cdot]_2)$ and $(\texttt{letrec}\ x = [\cdot]_2\ \texttt{in}\ [\cdot]_1)$, where the indices are used to denote the one-to-one-correspondences.

4.3 Space-Safety of Transformations

In this section the space behavior of several transformations is analyzed. Definitions of extra transformation rules are directly transferred from Definition 3.9 and 3.11 to LR. The forking diagrams are based on the diagrams of the appendix of (SSS08b). The measure for induction is usually (μ_1, μ_2, μ_3), ordered lexicographically, where μ_1 is the number of *LCSC*-reductions of an expression s to a WHNF and μ_2 is μ_{lll} and μ_3 is the syntactical size.

First, we show the first requirement of the context lemma for the normal order reduction rules except of (cp):

Lemma 4.9 (Requirement $\leq_{\mathbb{C},\texttt{sizetgc}}$ for Normal Order Reduction Rules Except of (cp))
For the transformations $a \in \{\text{(lbeta)}, \text{(case)}, \text{(seq)}, \text{(lll)}\}$ and expressions $s \xrightarrow{a} t$, we have $s \geq_{\mathbb{C},\texttt{sizetgc}} t$.

Proof
We use Lemma 4.7 implicitly in the following, which implies that garbage collectable positions are transported by the reductions.

For (lbeta) the preconditions of Lemma 4.4 hold, since (lbeta) is correct. Note that the positions of the characteristic multicontexts of rule (lbeta) are switched.

For the variants of (case) the conditions hold. For (llet), (lapp), (lcase) and (lseq) we have to formalize these rules by an infinite number of rule formats, since the bindings must be explicit to satisfy the conditions of Lemma 4.4 – there are no surprises.

(seq) may delete a subexpression, but since all garbage collectable positions are eliminated this rule satisfies the preconditions.

Note that the preconditions of Lemma 4.4 are in general not satisfied for (cp), since the resulting expression is larger in size and in general this also holds after applying (tgc).

We also show the second requirement of the context lemma for the normal order reduction rules except of (cp):

Lemma 4.10 (Requirement on *sps* for Normal Order Reduction Rules Except of (cp))
For the rules $a \in \{(\text{lbeta}), (\text{case}), (\text{seq}), (\text{lll})\}$ with $s \xrightarrow{a} t$, executed at top, the inequation *sps*$(\mathbb{R}[s]) \geq$ *sps*$(\mathbb{R}[t])$ holds.

Proof
The reduction step is the first one in the normal-order reduction and the maximum is taken over all reduction steps, hence the inequation obviously holds.

4.3.1 Total Space Improvement Properties of (lll), (seq-c) and (case-c)

In this section we show that (lll), (seq-c) and (case-c) are total space improvements.

Theorem 4.2 (Total Space Improvement Property of (llet))
The transformation (llet) is a total space improvement.

Proof
A complete set of forking diagrams in surfaces contexts is:

$$\begin{tikzcd}
\cdot \arrow[r, "{\mathbb{S},llet}"] \arrow[d, "{LR,a}"'] & \cdot \arrow[d, dashed, "{LR,a}"] \\
\cdot \arrow[r, dashed, "{\mathbb{S},llet}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "{\mathbb{S},llet}"] \arrow[d, "{LR,a}"'] & \cdot \arrow[dl, dashed, "{LR,a}"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "{\mathbb{S},llet}"] \arrow[d, "{LR,lll,+}"'] & \cdot \arrow[dl, dashed, "{LR,lll,+}"] \\
\cdot &
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[r, "{\mathbb{S},llet}"] \arrow[d, "{LR,lll,+}"'] & \cdot \arrow[d, dashed, "{LR,lll,+}"] \\
\cdot \arrow[r, dashed, "{\mathbb{S},llet}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "{\mathbb{S},llet}"] \arrow[d, "{LR,a}"'] & \cdot \arrow[ddl, dashed, "{LR,a}"] \\
\cdot \arrow[d, "{LR,llet}"'] & \\
\cdot &
\end{tikzcd}$$

Since (llet) only moves let-bindings, (llet) does not change the size and all garbage positions can be transferred directly, thus induction using the diagrams and applying Lemma 4.9, Lemma 4.10, Lemma 4.5 and Proposition 4.1 shows the claim.

For the following transformations only an analysis of reduction contexts is needed.

Theorem 4.3 (Total Space Improvement Property of (seq-c), (case-c), (lbeta), (lapp), (lcase), (lseq))
The transformations (seq-c), (case-c), (lbeta), (lapp), (lcase) and (lseq) are total space improvements.

Proof
By considering reduction contexts we see that each of the above reductions is a normal order reduction. Also the garbage collectable positions remain unchanged for each transformation, hence we apply Lemma 4.9, Lemma 4.10 and Lemma 4.5 to show that these transformations are total space improvements.

Using the last two theorems it is easy to show, that (lll) is a total space improvement.

Theorem 4.4 (Total Space Improvement Property of (lll))
The transformation (lll) is a total space improvement.

Proof
Follows from Theorem 4.2 and Theorem 4.3.

4.3.2 Total Space Equivalence Property of (cpx)

(cpx) is often used by other proofs, since the transformation is able to model subparts of other transformations and is a total space equivalence:

Theorem 4.5 (Total Space Equivalence Property of (cpx))
The transformation (cpx) is a total space equivalence.

Proof
An analysis of forking overlaps between LR-reductions and (cpx)-transformations in top contexts shows that the following diagram is complete, where all concrete (cpx)-transformations in a diagram copy from the same binding $x = y$:

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T},cpx"] \arrow[d, "LR{,}a"'] & s' \arrow[d, dashed, "LR{,}a"] \\
s_1 \arrow[r, dashed, "\mathbb{T},cpx,*"'] & s'_1
\end{tikzcd}$$

Let $s \xrightarrow{cpx} s'$. By induction on the number of LR-reductions of $\mathbb{T}[s]$ the equality $sps(\mathbb{T}[s]) = sps(\mathbb{T}[s'])$ holds.

We now show the requirements of the context lemma: s' might have a garbage `letrec`-binding from a variable to variable in contrast to s, without impact on `sizetgc` since variables are not counted by the `size`-measure. Using the diagram we see that $s \sim_{\mathbb{T},sps} s'$. Since $s \xrightarrow{cpx} s'$ does not introduce garbage that has an impact on `sizetgc` also $s \sim_{\mathbb{C},\texttt{sizetgc}} s'$ holds.

Since (cpx) does not affect the `size` and also does not have an impact on `sizetgc`, we have $\texttt{sizetgc}(s) = \texttt{sizetgc}(s')$ and then induction using the diagram above and application of Lemma 4.6 and Proposition 4.1 finishes the proof.

4.3.3 Total Space Equivalence Property of (xch)

In this section we show that (xch) is total space equivalence.

Theorem 4.6 (Total Space Equivalence Property of (xch))
The transformation (xch) is a total space equivalence.

Proof
An analysis of forking overlaps between LR-reductions and (xch)-transformations

in surface contexts shows that the following set of diagrams is complete:

$$\begin{tikzcd}
\cdot \arrow[r, "\mathbb{S}{,}xch"] \arrow[d, "LR{,}a"'] & \cdot \arrow[d, dashed, "LR{,}a"] \\
\cdot \arrow[r, dashed, "\mathbb{S}{,}xch"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{S}{,}xch"] \arrow[d, "LR{,}a"'] & \cdot \arrow[dl, dashed, "LR{,}a"] \\
\cdot &
\end{tikzcd}$$

We use induction using the diagrams to show that `size` and `sizetgc` is not changed by (xch).

Since (xch) only performs a renaming of `letrec`-variables, the garbage collection positions can be transferred directly, hence the context lemma can be applied.

Thus applying Lemma 4.5 and Proposition 4.1 shows that (xch) is a total space equivalence.

4.3.4 Total Space Equivalence Properties of (abs) and (abse)

In this section we show that (abs) and (abse) are total space equivalences.

Theorem 4.7 (Total Space Equivalence Properties of (abs) and (abse))
The transformations (abs) and (abse) are total space equivalences.

Proof
An analysis of forking overlaps between LR-reductions and (abs)-transformations in surface contexts shows that the following set of diagrams is complete:

$$\begin{tikzcd}
\cdot \arrow[r, "\mathbb{S}{,}abs"] \arrow[d, "LR{,}a"'] & \cdot \arrow[d, dashed, "LR{,}a"] \\
\cdot \arrow[r, dashed, "\mathbb{S}{,}abs"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{S}{,}abs"] \arrow[d, "LR{,}a"'] & \cdot \arrow[dl, dashed, "LR{,}a"] \\
\cdot &
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[rrr, "\mathbb{S}{,}abs"] \arrow[d, "LR{,}case"'] & & & \cdot \arrow[d, dashed, "LR{,}case"] \\
\cdot \arrow[r, dashed, "\mathbb{S}{,}abs"'] & \cdot \arrow[r, dashed, "\mathbb{S}{,}cpx{,}*"'] & \cdot \arrow[r, dashed, "\mathbb{S}{,}xch{,}*"'] & \cdot
\end{tikzcd}$$

We apply induction using the diagrams. Theorem 4.5 and Theorem 4.6 show that (cpx) and (xch) are total space equivalences, needed for the last diagram. Hence the `size` and `sizetgc` is not changed.

Since also garbage positions remain unchanged, all requirements are fulfilled to apply the context lemma. Thus an application of Lemma 4.6 and Proposition 4.1 shows that (abs) is a total space equivalence.

The proof for (abse) is analogous using the context lemma and the same total space equivalences.

4.3.5 Total Space Improvement Property of (seq)

In this section we show that (seq) is a total space improvement.

Theorem 4.8 (Total Space Improvement Property of (seq))
The transformation (seq) is a total space improvement.

Proof
Theorem 4.3 shows that (seq-c) is a total space improvement. In the other cases we analyze the forking overlaps between LR-reductions and (seq)-transformations in top contexts:

$$\begin{tikzcd}
\cdot \arrow[r, "\mathbb{T}{,}seq"] \arrow[d, "LR{,}a"'] & \cdot \arrow[d, dashed, "LR{,}a"] \\
\cdot \arrow[r, dashed, "\mathbb{T}{,}seq"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{T}{,}seq"] \arrow[d, "LR{,}a"'] & \cdot \arrow[ddl, dashed, "LR{,}a"] \\
\cdot \arrow[d, "LR{,}seq"'] & \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{T}{,}seq"] \arrow[d, "LR{,}a"'] & \cdot \arrow[dl, dashed, "LR{,}a"] \\
\cdot &
\end{tikzcd}$$

(seq) does not increase `size` and also not `sizetgc`, hence induction using the diagrams and applying Lemma 4.9, Lemma 4.10 and Lemma 4.5 finishes the proof.

4.3.6 Total Space Improvement Properties of (case) and (case*)

In this section show that (case) and (case*) are total space improvements. We start with (case):

Theorem 4.9 (Total Space Improvement Property of (case))
The transformation (case) is a total space improvement.

Proof
Theorem 4.3 shows that (case-c) is a total space improvement. In the other cases we analyze the forking overlaps between LR-reductions and (case)-transformations in top contexts:

$$\begin{tikzcd}
\cdot \arrow[r, "\mathbb{T}{,}case"] \arrow[d, "LR{,}a"'] & \cdot \arrow[d, dashed, "LR{,}a"] \\
\cdot \arrow[r, dashed, "\mathbb{T}{,}case"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{T}{,}case"] \arrow[d, "LR{,}a"'] & \cdot \arrow[ddl, dashed, "LR{,}a"] \\
\cdot \arrow[d, "LR{,}case"'] & \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{T}{,}case"] \arrow[d, "LR{,}a"'] & \cdot \arrow[dl, dashed, "LR{,}a"] \\
\cdot &
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[rrr, "\mathbb{T}{,}case"] \arrow[d, "LR{,}case"'] & & & \cdot \arrow[d, dashed, "LR{,}case"] \\
\cdot \arrow[r, dashed, "\mathbb{T}{,}case"'] & \cdot \arrow[r, dashed, "cpx{,}*"'] & \cdot \arrow[r, dashed, "xch{,}*"'] & \cdot
\end{tikzcd}$$

(case) does not increase `size` and also not `sizetgc`. Hence we apply induction using the diagrams, where Theorem 4.5 and Theorem 4.6 is needed for the last diagram. The application of Lemma 4.9, Lemma 4.10 and Lemma 4.5 finishes the proof.

Using the total space improvement property of (case) we now can show that (case*) is also a total space improvement.

Theorem 4.10 (Total Space Improvement Property of (case*))
The transformation (case*) is a total space improvement.

Proof
We use the transformation (case-cx), that is defined in Definition 3.11, where also the definition of (case*) is given.

In (SSS08b) (case-cx) is simulated using (case) and a subsequent $\xrightarrow{cpx,*}\xrightarrow{gc,*}$ and using Theorem 4.9, Theorem 4.5, the correctness of (gc) and Theorem 4.1 this shows that (case-cx) is a total space equivalence. Since (case*) is either (case) or (case-cx) per definition, (case*) is a total space improvement.

4.3.7 Total Space Equivalence Property of (ucp)

The transformation (ucp) is also a total space equivalence:

Theorem 4.11 (Total Space Equivalence Property of (ucp))
The transformation (ucp) is a total space equivalence.

Proof
An analysis of forking overlaps between LR-reductions and (ucp)-transformations in surface contexts shows that the following set of diagrams is complete:

$$\begin{tikzcd}
\cdot \arrow[r, "\mathbb{S},ucp"] \arrow[d, "LR,a"'] & \cdot \arrow[d, dashed, "LR,a"] \\
\cdot \arrow[r, dashed, "\mathbb{S},ucp"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{S},ucp"] \arrow[d, "LR,a"'] & \cdot \arrow[dl, dashed, "LR,a"] \\
\cdot &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{S},ucp"] \arrow[d, "{LR,lll,+}"'] & \cdot \arrow[d, dashed, "{LR,lll,*}"] \\
\cdot \arrow[r, dashed, "\mathbb{S},ucp"'] & \cdot
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[r, "\mathbb{S},ucp"] \arrow[d, "LR,cp"'] & \cdot \\
\cdot \arrow[ur, dashed, "\mathbb{S},gc"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathbb{S},ucp"] \arrow[d, "LR,a"'] & \cdot \arrow[d, dashed, "LR,a"] \\
\cdot \arrow[r, dashed, "\mathbb{S},gc"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[rrr, "\mathbb{S},ucp"] \arrow[d, "LR,case"'] & & & \cdot \arrow[d, dashed, "LR,case"] \\
\cdot \arrow[r, dashed, "\mathbb{S},gc"'] & \cdot \arrow[r, dashed, "{\mathbb{S},cpx,*}"'] & \cdot \arrow[r, dashed, "{\mathbb{S},gc,*}"'] & \cdot
\end{tikzcd}$$

(gc) is a total space improvement, which follows from the correctness of (gc) and Theorem 4.1.

Induction using the diagrams, where also Theorem 4.5 is needed, shows that `size` and also `sizetgc` remain unchanged.

Since also garbage collectable positions are unchanged Lemma 4.6 and Proposition 4.1 shows that (ucp) is a total space equivalence.

4.3.8 Space-Properties of (cp), (cpS) and (cpcx)

None of the transformations (cp), (cpS) and (cpcx) are total space improvements. This means that also using total garbage collection, the normal order reduction is not an overall space improvement.

Theorem 4.12 (Space-Properties of (cp), (cpS) and (cpcx))
The transformations (cp), (cpS) and (cpcx) are in general no total space improvements.

Proof
It is sufficient to present a counterexample. Consider the following expression that uses tuples in Haskell-notation:

$$\texttt{letrec}\ x = (\texttt{True}, \texttt{False})\ \texttt{in}\ \texttt{seq}\ x\ (\texttt{seq}\ x\ x)$$

A normal-order reduction sequence is as follows:

$$\begin{array}{rl} & \texttt{letrec}\ x = (\texttt{True}, \texttt{False})\ \texttt{in}\ \texttt{seq}\ x\ (\texttt{seq}\ x\ x) \\ \xrightarrow{LR,seq-in} & \texttt{letrec}\ x = (\texttt{True}, \texttt{False})\ \texttt{in}\ \texttt{seq}\ x\ x \\ \xrightarrow{LR,seq-in} & \texttt{letrec}\ x = (\texttt{True}, \texttt{False})\ \texttt{in}\ x \end{array}$$

In contrast, copying results in

$$\texttt{letrec}\ x = (\texttt{True}, \texttt{False})\ \texttt{in}\ \texttt{seq}\ x\ (\texttt{seq}\ (\texttt{True}, \texttt{False})\ x)$$

which has an *sps* that is strictly greater than before. Note that we have to apply (tgc) before measuring.

If we use (cpcx) instead, then the corresponding expression is the following, that also has a strictly greater *sps* than before:

$$\texttt{letrec}\ x = (y_1, y_2), y_1 = \texttt{True}, y_2 = \texttt{False}\ \texttt{in}\ \texttt{seq}\ x\ (\texttt{seq}\ (y_1, y_2)\ x)$$

However, applied in normal-order, (cp) is an opportunistic total space improvement:

Proposition 4.2 (Opportunistic Total Space Improvement Property of (cp) in Normal Order)
The reduction (cp) applied in normal-order is an opportunistic total space improvement.

Proof
This holds, since *sps* maximizes the size values along the normal-order reduction sequence, the transformation is at the start of it and no contexts are involved.

We also give an analysis of the maximal space increase of (cpS) in surface contexts:

Proposition 4.3 (Space-Property of (cpS) in Surface Contexts)
The transformation $(\mathbb{S}, cpS)$ increases the maximal space by at most $\texttt{size}(v)$, where v is the copied abstraction.

Proof
Let $s \xrightarrow{\mathbb{S},cpS} s'$, where v is the copied abstraction.

A complete set of forking diagrams for (cpS) in surface contexts is as follows:

$$\begin{tikzcd}
s \arrow[r, "\mathbb{S},cpS"] \arrow[d, "LR,a"'] & s' \arrow[d, dashed, "LR,a"] \\
s_1 \arrow[r, dashed, "\mathbb{S},cpS"'] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{S},cpS"] \arrow[d, "LR,a"'] & s' \arrow[ddl, dashed, "LR,a"] \\
s_1 \arrow[d, "LR,cpS"'] & \\
s_2 &
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{S},cpS"] \arrow[d, "LR,a"'] & s' \arrow[dl, dashed, "LR,a"] \\
s_1 &
\end{tikzcd}$$

We show the claim using induction:

If s is a LR-WHNF, then s' is also a LR-WHNF and $\texttt{size}(s) = \texttt{size}(v) = \texttt{size}(s')$ holds, thus we obtain the claim.

Now s has a LRPgc-reduction and we go through the diagrams. If diagram 1 applies, then the induction hypothesis applies to s_1 and $\texttt{size}(s) + \texttt{size}(v) = \texttt{size}(s')$ holds, hence $\mathit{sps}(s') \leq \mathit{sps}(s) + \texttt{size}(v)$. If diagram 2 or 3 applies, then the computation is similar as above.

We also analyze (cpcxT) applied in top contexts:

Proposition 4.4 (Space-Property of (cpcxT) in Top Contexts)
If $s \xrightarrow{\mathbb{T},cpcxT} s'$, then $\mathit{sps}(s) \leq \mathit{sps}(s') \leq \mathit{sps}(s) + 1$.

Proof
The inequality is shown by an induction argument using the following diagrams in top contexts:

$$\begin{tikzcd}
s \arrow[r, "\mathbb{T},cpcxT"] \arrow[d, "LR,a"'] & s' \arrow[d, dashed, "LR,a"] \\
s_1 \arrow[r, dashed, "\mathbb{T},cpcxT,0\vee 1"] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s \arrow[r, "\mathbb{T},cpcxT"] \arrow[d, "LR,a"'] & s' \arrow[d, dashed, "LR,a"] \\
s_1 \arrow[r, dashed, "\mathbb{T},abs"] & s'_1
\end{tikzcd}$$

$a \in \{seq, case\}$

$$\begin{tikzcd}
s \arrow[rrr, "\mathbb{T},cpcxT"] \arrow[d, "LR,case"'] & & & s' \arrow[d, dashed, "LR,case"] \\
s_1 \arrow[r, dashed, "\mathbb{T},cpcxT"'] & s_2 \arrow[r, dashed, leftrightarrow, "\mathbb{T},cpx,*"'] & \cdot \arrow[r, dashed, leftrightarrow, "\mathbb{T},gc=,*"'] & s'_1
\end{tikzcd}$$

The transformation (cpcx) does not change the LR-WHNF property.

We show the claim by induction using the diagrams above.

If s, s' are LR-WHNFs, then the claim holds, since $\texttt{size}(s) + 1 = \texttt{size}(s')$. In all other cases s is LR-reducible and we have to check the diagrams:

Assume the first diagram is applicable, then we have $\mathit{sps}(s') = \mathit{sps}(s) + 1$ using the induction hypothesis.

In the case of diagram 2 we obtain *sps*(s_1) = *sps*(s'_1) by Theorem 4.7 and since $\texttt{size}(s') = \texttt{size}(s) + 1$, this implies the claim.

In the third diagram, the induction hypothesis can be applied. This together with Theorem 4.5, the correctness of (gc=) and Theorem 4.1 shows *sps*(s_2) = *sps*(s'_1) and *sps*$(s_1) \leq$ *sps*$(s_2) \leq$ *sps*$(s_1) + 1$.

4.3.9 Space-Properties of (cse) and (soec)

In this section we analyze the space behavior of (cse) and (soec) (see Definition 3.12 for the definition of both functions in LRP, directly portable to LR). Both transformations might increase the space consumption by an arbitrary amount, hence they are space leaks if we use the same term as defined in Definition 3.8 ported to LR.

Proposition 4.5 ((cse) is a Space Leak)
The transformation (cse) is a space leak.

Proof
This follows from the proof of Proposition 3.7, but since another garbage collection approach is used here, we present the proof shortened:

The expression s is given in a Haskell-like notation, using integers, but can also be defined in LR:

$$\texttt{if}\ (\texttt{last}\ [1..n]) > 0\ \texttt{then}\ [1..n]\ \texttt{else}\ \texttt{Nil}$$

where $[1..n]$ lazily generates a list $[1, \ldots, n]$ and `last` returns the last element of a list, i.e. forcing tail-strictness. Thus both lists $[1..n]$ are calculated twice, where the calculation of the second stops after evaluating the start of the list and requires constant space. During evaluation of the first $[1..n]$, all lists element are directly garbage collected after creation, hence only constant space is needed.

If we have $s \xrightarrow{cse} s'$, then s' is:

$$s' = \texttt{letrec}\ x = [1..n]\ \texttt{in}\ (\texttt{if}\ (\texttt{last}\ x) > 0\ \texttt{then}\ x\ \texttt{else}\ \texttt{Nil})$$

Now even total garbage collection cannot remove the binding x, since x is needed later and the list is stored in full length in memory. The size required is a linear function in n, where instead of n also $f(n)$ could be used for any computable function f, causing an unbound space increase.

The same holds for (soec):

Proposition 4.6 ((soec) is a Space Leak)
The transformation (soec) is a space leak.

Proof
As already in the proof of Proposition 3.8, this follows from (BR00).

4.3.10 Summary

In this section we summarize the results of the previous sections.

The normal order reduction is a total space improvement except of (cp):

Theorem 4.13 (Space-Property of Normal Order Reduction in LR)
The normal order reduction of LR is a total space improvement except of (cp).

Proof
This follows from Theorem 4.3, Theorem 4.4, Theorem 4.8 and Theorem 4.9.

Finally we give an overview of all results of this section:

Theorem 4.14 (Space-Properties of Several LR-Transformations)
The following table shows the space properties of all transformations analyzed in this section:

Space-Property	Transformations
$\leq_{spmax}$	(lbeta), (seq), (case), (lll), (gc), (case*)
$\sim_{spmax}$	(cpx), (abs), (abse), (xch), (ucp), (case-cx), (gc=)
$\not\leq_{spmax}$	(cpcx), (cpS), (tgc)
space-safe up to 1	($\mathbb{T}$,cpcxT)
space-safe up to $\texttt{size}(v)$ where v is the copied abstraction	($\mathbb{S}$,cpS)
space-leak	(cp), (cse), (soec)

Proof
This follows from the correctness of (gc) and (gc=) together with Theorem 4.1, Theorem 4.5, Theorem 4.6, Theorem 4.7, Theorem 4.10, Theorem 4.11, Theorem 4.12, Theorem 4.13, Proposition 4.3, Proposition 4.4, Proposition 4.5 and Proposition 4.6.

4.4 Classification of Total Garbage Collection

In general total garbage collection detects unnecessary subexpressions. However this kind of garbage collection is based on the transformation (tgc) and is still not optimal. Consider the following program:

$$\texttt{letrec}\ x = \ldots\ \texttt{in}\ \mathbb{C}[\texttt{seq}\ x\ x]$$

Depending on the context $\mathbb{C}$ an optimal garbage collector is able to remove the `seq`-expression, resulting in `letrec` $x = \ldots$ `in` $\mathbb{C}[x]$, if the semantics of the program are not changed. Using (tgc) this is not possible, since positions need to be replaced by Bot and this is not possible in our example.

However an optimal garbage collector has some pitfalls: Often it might be possible to remove some subexpressions as in the example above, but especially `seq`-expressions can be used to force evaluations and to improve the runtime or space-behavior in some situations. Hence if we want to define an optimal garbage collector, then we need to be aware that it may not worsen the runtime or space consumption only by removing a subexpression. While (tgc) is based on program correctness, using the criteria of convergence, such an optimal garbage collector is much more complex.

Especially if we compare the total garbage collection with the deterministic eager garbage collection used in Chapter 3, then we see that in both cases (cp) is a space leak. To make the counterexample easy to grasp, we used an approach that really exploits the fact, that (tgc) needs to evaluate unneeded `seq`-expressions (compare the proof of Theorem 4.12). But also using an optimal garbage collector, we can find an example that shows, that (cp) is a space leak:

Proposition 4.7 ((cp) is a Space Leak using an Optimal Garbage Collector)
The transformation (cp) is a space leak, even if an optimal garbage collector is used.

Proof
Consider a large program containing an also large function f. This function f is used two-times during the normal-order evaluation of the program, the first time after t_1 normal-order-steps, where p_1 describes the target-position of the (cp), the second time after t_2 normal-order-steps, where p_2 describes the target position. Now we assume $t_1 \ll t_2$ and also that t_1 is a great number.

Until t_2 is reached, the function f cannot be garbage collected, since the reference of f exists for any $t < t_2$.

Now we apply a non-normal-order (cp), copying f to p_1 after t_0 normal-order-steps, where t_0 is a small number, hence $t_0 \ll t_1 \ll t_2$. Then we proceed using normal-order. Since the whole subprogram at p_1 needs this copy of f, the function-copy cannot be removed there. But also at the original definition-site f cannot be garbage collected, since f is still referenced by p_2. Hence, until the normal-order-evaluation reaches p_1 we have an additional space consumption of $\texttt{size}(f)$ compared to the normal-order-reduction.

Since f can be scaled without changing the semantics, this example shows that (cp) causes a space leak, even if an optimal garbage collector is used.

As already mentioned in the introduction of this section, we introduced total garbage collection, so that we are able to show that certain transformations are space improvements, space equivalences or space leaks independent of the concrete implementation of garbage collection.

It is not clear, whether a total space improvement property of a transformation implies that it is also a maximal space improvement (see Chapter 3) or vice versa. This is caused by the different methods of garbage collection and confirms the motivation of a total garbage collector. Total garbage collection is a good reference for the space-safeness of program transformations and the results for the transformations are almost independent of garbage collection.

The comparison of Section 4.3.10 and Section 3.4.14 shows that eager garbage collection is at least a close approximation of total garbage collection. Seen from the other direction, the results of a calculus using an practically unimplementable total garbage collector can be used carefully for a calculus using a practical garbage collector.

5

Runtime-Optimizations in CHF

In the deterministic scenario a lot of work in the area of time improvements is already done, e.g. (MS99, HH14, SS15, SSS15a). In (BFKT00, THLP98, MNJ11) parallelism is introduced into deterministic programs. Improvements in a nondeterministic scenario are studied in (LM99) using a call-by-name calculus with McCarthy's amb-operator.

(SSSD18) considers time improvements using a concurrent programming language with shared memory and side-effecting computations, where CHF* is used together with the usual single-processor model for sequential and a multi-processor model for parallel evaluation.

Many fundamentals needed for time analyses are also used for space analyses. Therefore the work of (SSSD18) is summarized in this chapter.

5.1 Sequential and Parallel Time Improvements

In deterministic programming languages, a program transformation is a time improvement iff it is correct and it does not increase the length of reduction sequences in any context (e.g. see (SSS16b)). The length of reduction sequences may take into account only essential reductions steps instead of all steps.

In contrast to deterministic languages, the concurrency leads to a nondeterministic evaluation with potentially different results and also to a potential speed up caused by the concurrent threads if multiple processors are available.

Two improvement relations are considered, where the first one can be seen as a single-processor model, while the other one is adapted for a multi-processor scenario.

Sequential reductions are considered for a sequence of interleaved reductions of concurrent threads and *parallel reductions* where threads run in parallel. The definition of time improvement in CHF* is parametrized, following the approach of (SSS17), where only a subset of reduction rules are used for measuring the length of sequences. These subsets are defined using sets of so-called *reduction kinds*:

- A is a set of all rule-names of Definition 2.21.
- A_{all} is the set of all reduction kinds.
- $A_{cp} := \{(\text{cp}), (\text{cpcxa}), (\text{cpcxb}), (\text{mkbinds})\}$ and $A_{noncp} := A_{all} \backslash A_{cp}$

For time analyses A_{noncp} is sufficient: Abstract machines as CIOM1 do not execute (cp) for variables, nor (cpcxa), i.e. copying variables and abstracting functional or monadic values, caused by the restricted structure of machine expressions. The other reduction kinds (cp), (cpcxb) and (mkbinds) only occur as follows, where reduction steps are viewed per thread: Several (mkbinds) are always followed by an A_{noncp}-reduction or by a (cp) which copies an abstraction and then an A_{noncp}-reduction. The same holds for (cpcxb). The additional effort per A_{noncp}-step is at most the size of the initial process.

A sequential A-improvement improves the length of minimal and successful reduction sequences w.r.t. the reduction kinds in A. In the case of non-terminating reduction sequences, taking the minimum as defined for $srnr_A(\cdot)$ automatically prefers finite reduction sequence, if some exists.

Definition 5.1 (CHF* Sequential Time Improvement and Equivalence)
Let P be a well-formed process with $P\downarrow$, $A \subseteq A_{all}$ and *Red* be a successful reduction sequence of P. Let $srnr_A(\mathit{Red})$ be the number of A-reductions occurring in Red. We define $srnr_A(P) := \min\{srnr_A(Red) \mid Red$ is a successful standard reduction of $P\}$.
Let P_1 and P_2 be two well-formed processes with $P_1\downarrow, P_2\downarrow$ and $P_1 \sim_c P_2$.
If $srnr_A(\mathbb{D}[P_1]) \leq srnr_A(\mathbb{D}[P_2])$ holds for all $\mathbb{D} \in$ *PCtxt*, then P_1 *sequentially A-improves* P_2, written $P_1 \preceq_A P_2$. If $P_1 \preceq_A P_2$ and $P_2 \preceq_A P_1$, then we say P_1, P_2 are *improvement-equivalent w.r.t. A.*
A program transformation $\xrightarrow{PT}$ *is a sequential A-improvement* if $P_1 \xrightarrow{PT} P_2$ implies that P_2 sequentially A-improves P_1 for all processes P_1, P_2.
We say that $\xrightarrow{PT}$ is a *sequential A-improvement equivalence* iff $\xrightarrow{PT}$ and $\xrightarrow{PT^-}$ (the inverse of $\xrightarrow{PT}$) are both sequential A-improvements.

Parallel evaluation is defined as follows:

Definition 5.2 (Parallel Evaluation)
Let P be a well-formed process and assume w.l.o.g. that it is in ν-prenex form $\nu x_1 \dots x_m.P_0$. We write P_0 using a multicontext $M[\cdot_1, \dots, \cdot_n] \mid [\cdot_{n+1}] \dots \mid [\cdot_{2n}]$ with $2n$ holes such that $P_0 = M[e_1, \dots, e_n] \mid Q_1 \dots \mid Q_n$ where e_i are expressions or threads and Q_i are processes. Let V_i be a set of variables, Q_i' be processes. Then a *parallel reduction* is defined as follows

$$\begin{aligned} &\nu x_1 \dots x_m.M[e_1, \dots, e_n] \mid Q_1 \mid \dots \mid Q_n \\ \to\ &\nu x_1 \dots x_m.\nu V_1, \dots V_n.M[e_1', \dots, e_n'] \mid Q_1' \mid \dots \mid Q_n' \end{aligned}$$

provided the following holds:

1. For every i:
   $\nu x_1 \dots x_m.M[e_1, \dots, e_n] \mid Q_1 \mid \dots \mid \mid Q_n \xrightarrow{sr}$
   $\nu x_1 \dots x_m.\nu V_i.M[e_1, \dots, e_{i-1}, e_i', e_{i+1}, \dots, e_n] \mid Q_1 \mid \dots \mid Q_{i-1} \mid Q_i' \mid Q_{i+1} \dots Q_n$
   is a single reduction where e_i is the redex of the reduction (note that for rule (sr,unIO), e_i is a thread and not an expression).
2. The result is a valid process and $i > 0$.
3. The Processes Q_i and Q_i' and the sets of variables V_i may be empty.

A parallel reduction sequence is *successful* if the last process is successful. The number of parallel single reductions in a parallel reduction step is limited by the number of available processors, sometimes denoted by the number N.

A parallel improvement improves the length of minimal and also successful reduction sequences w.r.t. the reduction kinds in A and the number of processors N:

Definition 5.3 (CHF* Parallel Time Improvement and Equivalence)
Let P be a well-formed process with $P\downarrow$, let A be a set of reduction kinds and let $N \in \{1, 2, \dots\} \cup \{\infty\}$ be the number of available processors. For a parallel reduction sequence *Red*, let $\mathrm{srnrp}_A^N(\mathit{Red})$ be the number of parallel reduction steps for at most N processors that contain an A-reduction. If $N = \infty$, then we may omit the superscript N. Let $\mathrm{srnrp}_A^N(P)$, the *parallel number of A-steps*, be the minimum of:

$$\{\mathrm{srnrp}_A^N(\mathit{Red}) \mid \mathit{Red} \text{ is a successful parallel reduction with at most } N \text{ processors of } P\}$$

For well-formed P_1, P_2 with $P_1\downarrow, P_2\downarrow, P_1 \sim_c P_2$, we say P_1 *parallel improves* P_2 w.r.t. A and N processors, notation $P_1 \preceq_{p,N,A} P_2$, iff $\mathrm{srnrp}_A^N(\mathbb{D}[P_1]) \leq \mathrm{srnrp}_A^N(\mathbb{D}[P_2])$ holds for all $\mathbb{D} \in \mathit{PCtxt}$.
If $P_1 \preceq_{p,N,A} P_2$ and $P_2 \preceq_{p,N,A} P_1$, then we say P_1, P_2 are *improvement-equivalent w.r.t. A and N* and parallel reduction. A program transformation $\xrightarrow{PT}$ is a *parallel improvement w.r.t. A and N processors*, iff $P_2 \xrightarrow{PT} P_1$ implies $P_1 \preceq_{p,N,A} P_2$ for all processes P_1 and P_2 and it is a *parallel improvement equivalence w.r.t. A and N* iff $P_2 \xrightarrow{PT} P_1$ implies that P_1 and P_2 are improvement-equivalent w.r.t. A and N.

5.2 Proven Time Improvements

Before we give an overview of showed time improvements, several additional transformation rules are needed. We start with the following rules that cannot be derived from rules for LRP. The rules (dtmvar) and (dpmvar) are variants of (sr,tmvar) and (sr,pmvar) where the side conditions ensure that the MVar-access is deterministic. The rule (drfork) removes a future-operation and thus performs thread elimination provided that the corresponding computation does not access any MVar.

Definition 5.4 (Extra CHF*-Specific Transformation Rules)

(dtmvar) $\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\,\mathbf{m}\,e] \rightarrow \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return}\ e] \mid x\,\mathbf{m}\,-]$
if for all $\mathbb{D}' \in \mathit{PCtxt}$ and $\xrightarrow{sr,*}$-sequences starting with $\mathbb{D}'[\nu x.(\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\,\mathbf{m}\,e])]$ the first execution of any $(\texttt{takeMVar}\ x)$-operation takes place in the y-thread.

(dpmvar) $\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x\,\mathbf{m}\,-] \rightarrow \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return}\ ()] \mid x\,\mathbf{m}\,e]$
if for all $\mathbb{D}' \in \mathit{PCtxt}$ and $\xrightarrow{sr,*}$-sequences starting with $\mathbb{D}'[\nu x.(\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x\,\mathbf{m}\,-])]$ the first execution of any $(\texttt{putMVar}\ x\ e')$-operation takes place in the y-thread.

(drfork) $\mathbb{C}[\texttt{future}\ e] \rightarrow \mathbb{C}[e]$
if for all $\mathbb{D} \in \mathit{PCtxt}$ and $\xrightarrow{sr,*}$-sequences starting with $\mathbb{D}[\mathbb{C}[\texttt{future}\ e]]$ the threads, started with $\texttt{future}\ e$, never will execute an action on an MVar.

The following rules are generalizations of rules from LRP (compare Definition 3.9 and Definition 3.12).

Definition 5.5 (Extra CHF* Transformation Rules (ucp) and (cse))

(ucpt)	$\nu x.(\mathbb{S}[x] \mid x = e) \to (\mathbb{S}[e])$, if x does not occur in $\mathbb{S}, e$ and $\mathbb{S}$ does not bind x
(ucpt)	$\mathbb{S}_1[\texttt{letrec}\ x = e, E\ \texttt{in}\ \mathbb{S}_2[x]] \to \mathbb{S}_1[\texttt{letrec}\ E\ \texttt{in}\ \mathbb{S}_2[e]]$ if x does not occur elsewhere and $\mathbb{S}_1$ and $\mathbb{S}_2$ do not bind x
(ucpd)	$\mathbb{C}_1[\lambda y.\mathbb{C}_2[\texttt{letrec}\ x = e, E\ \texttt{in}\ \mathbb{S}[x]]] \to \mathbb{C}_1[\lambda y.\mathbb{C}_2[\texttt{letrec}\ E\ \texttt{in}\ \mathbb{S}[e]]]$ if x does not occur elsewhere and $\mathbb{C}_1, \mathbb{C}_2$ and $\mathbb{S}$ do not bind x
(ucp)	(ucpd) or (ucpt)
(cse)	$\mathbb{C}[e] \mid x = e \to \mathbb{C}[x] \mid x = e$
(cse)	$\mathbb{C}[\texttt{letrec}\ E\ \texttt{in}\ e] \mid E' \to \mathbb{C}[\pi{\cdot}e] \mid E'$ if $\pi{\cdot}E =_\alpha E'$ for some permutation π that maps $LV(E) \to LV(E')$ and $LV(E)$ is fresh for E'
(cse)	$x_1 = e_1 \mid \ldots \mid x_n = e_n \mid y_1 = e'_1 \mid \ldots \mid y_n = e'_n \to x_1 = \pi{\cdot}e_1 \mid \ldots \mid x_n = \pi{\cdot}e_n$ if $\pi{\cdot}e_i =_\alpha \pi{\cdot}e'_i$ for the permutation π with $\forall i : \pi(x_i) = y_i, \pi(y_i) = x_i$ the variables x_i are not free in e_j for all j.

The permutation π in (cse) is a variable-to-variable function on the expressions.

For a the inverse of a program transformation PT we write PT^-.

Now the time improvement results can be summarized in the following table:

Transformation	Sequential A-improvement	Parallel A-improvement for every N
(sr,a), $a \notin \{\text{tmvar}, \text{pmvar}\}$	$A \subseteq A_{all}$	$A \subseteq A_{all}$
(lbeta)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$
(case)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$
(seq)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$
(mkbinds)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$
(cp)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$
(gc)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$
(gc)$^-$	$A \subseteq A_{all} \backslash \{\text{mkbinds}\}$	$A \subseteq A_{noncp}$
(ucp)	$A \subseteq A_{all} \backslash \{\text{cpcxa}\}$	$A \subseteq A_{noncp}$
(ucp)$^-$	$A \subseteq A_{noncp}$	$A \subseteq A_{noncp}$
(cpcxa)	$A \subseteq A_{all} \backslash \{\text{mkbinds}\}$	$A \subseteq A_{noncp}$
(cpcxb)	$A \subseteq A_{all} \backslash \{\text{mkbinds}, \text{cpcxa}\}$	$A \subseteq A_{noncp}$
(cse)	$A \subseteq A_{noncp}$	$A \subseteq A_{noncp}$
(lunit)	$A \subseteq A_{noncp}$	$A \subseteq A_{noncp}$
(dtmvar)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$
(dpmvar)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$
(drfork)	$A \subseteq A_{noncp}$	–
(drfork)$^-$	–	$A \subseteq (A_{noncp} \backslash \{\text{fork}, \text{unIO}\})$

The A-improvement equivalence properties for (gc) and (ucp) follow from the table. Thus the translation σ from CHF*-processes to simplified processes (see Definition 2.29) is an improvement equivalence w.r.t. $A \subseteq A_{noncp}$.

6

Space-Optimizations in CHF

The goal of this chapter is to analyze the space behavior of concurrent lazy-evaluating functional programs using a similar approach as used in Chapter 3.

As calculus CHF* (see Section 2.2) is used, where a space measurement is needed and defined in the next section. Since CHF* is nondeterministic, there might be many different evaluation paths for the same program. But even if the result is the same, the schedules have a great impact on the space consumption. Therefore we develop an efficient algorithm that computes a space-optimal schedule, if all threads run independently. We also give algorithms for threads with dependencies. Moreover an environment for space analyses in CHF* is developed.

6.1 Space Measurement and Garbage Collection

In this section a space measurement for CHF* is defined. It is crucial to recognize garbage and remove it, since we are interested in space improving transformations and not in garbage. This section is based on (SSD19a).

First of all we define the size of expressions and processes, that is a generalization of the `size`-measure for LRP (compare Definition 3.5).

Definition 6.1 (CHF* Size of Programs)
The `size` of CHF*-programs is defined as follows:

$$\begin{aligned}
&\mathtt{size}(x) &&= 0\\
&\mathtt{size}(s\ t) &&= 1 + \mathtt{size}(s) + \mathtt{size}(t)\\
&\mathtt{size}(\lambda x.s) &&= 1 + \mathtt{size}(s)\\
&\mathtt{size}(\mathtt{case}\ s\ \mathtt{of}\ \mathtt{alt}_1 \ldots \mathtt{alt}_n) &&= 1 + \mathtt{size}(s) + \textstyle\sum_{i=1}^{n} \mathtt{size}(\mathtt{alt}_i)\\
&\mathtt{size}((c\ x_1 \ldots x_n)\ \rightarrow\ s) &&= 1 + \mathtt{size}(s)\\
&\mathtt{size}(f\ s_1 \ldots s_n) &&= 1 + \textstyle\sum \mathtt{size}(s_i) \quad \text{for constructors and operators}\\
& && \qquad\qquad\qquad\quad f \text{ such as } \mathtt{future}, \mathtt{return}, \ldots\\
&\mathtt{size}(\mathtt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \mathtt{in}\ s) &&= \mathtt{size}(s) + \textstyle\sum \mathtt{size}(s_i)\\
&\mathtt{size}(P_1 \mid P_2) &&= \mathtt{size}(P_1) + \mathtt{size}(P_2)\\
&\mathtt{size}(x\ op\ s) &&= \mathtt{size}(s) \quad \text{for } op \in \{=, \mathbf{m}\}\\
&\mathtt{size}(x \Leftarrow s) &&= 1 + \mathtt{size}(s)\\
&\mathtt{size}(x\,\mathbf{m}\,-) &&= 0\\
&\mathtt{size}(\nu x.P) &&= 1 + \mathtt{size}(P)
\end{aligned}$$

We restrict the garbage collection to the components on top of the whole program.

Definition 6.2 (Garbage Collection Transformation Rule for CHF*)
Garbage collection is defined by the transformation

$$P_1 \xrightarrow{gc} P_2$$

where P_2 is generated from P_1, such that a maximal set M of the following components and bindings is removed:

- $x = e$
- $x\,\mathbf{m}\,e$
- $x\,\mathbf{m}\,-$

such that the following conditions hold (where occurence means proper occurrence and not only in the binder νy):

1. The variables x in top-level sharing, threads and MVars are ν-bound or bound by a binding.
2. The variables x do not occur as ν-bound or free variables in P_2.
3. There is no variable y that occurs in P_2 and which is also free in M.

After this operation also empty `letrec`-environments and νx-binding operators that do not bind any x are removed.

Based on the measure `size`, a space measure for evaluation sequences is defined as follows:

Definition 6.3 (CHF* Space Measures *sps* and *spmin*)
Let `sizegc` be the size of an program P after (gc) was applied.
Then the space measure *sps*(Red) of a successful standard reduction Red of a program P is the maximum of all `sizegc`(P_i) during the whole standard reduction sequence, $Red = P \xrightarrow{sr} P_1 \xrightarrow{sr} \dots \xrightarrow{sr} P_n$.
The space measure of a CHF*-program P is defined as follows:

$$spmin(P) = \min\{sps(Red) \mid Red \text{ is a successful standard reduction of } P\}$$

As also for LRP it is important that we only consider `size`-values that are free of garbage, because otherwise the results would be distorted by bindings that can be directly garbage collected after creation.

6.2 Space-Optimal Schedules

A first step before we can give a useful definition of space improvements for CHF* is to find a space-optimal schedule for a CHF*-program. Caused by the nondeterminism introduced by concurrency, there might be many different schedules for a single program. Also the relation between runtime and space behavior is not trivial and it might occur that a program with a good space consumption has a bad runtime behavior and vice versa.

However in practice we often do not have that many concurrent processes and sometimes those processes also have only rare interactions by a controllable form of synchronization. Therefore we first develop an algorithm that computes a space-minimal execution sequence of a set of given parallel and independent processes.

Afterwards also processes with interactions are analyzed. Note that processes can be implemented as threads in CHF*, but we use the term processes, since the algorithms in this section are more general and not restricted to CHF*.

Even if the processes are independent from each other, they still use a common memory and are stored in this common memory, but each process has its completely own areas in this memory. Also the state of processes are stored in this common memory. The algorithm for independent processes takes a list of numbers for each process that represents the space consumption at the corresponding time point, where the time point is equal to the index of the list-item. The abstract model can be applied if all processes have a common start and end time.

If we consider only two processes, then we have two of such lists and even two of such lists lead to an exponential number of possible schedules. Hence a brute force approach is to naive and we need a better algorithm. Also note that such an algorithm works offline, since the whole sequence of such numbers is required as prerequisite. Because the processes run independently, such lists of numbers can be calculated in a straightforward way.

The simplicity of this model allows applications of the space-optimization algorithm in the following scenarios:

- Industrial processes (jobs) where the number of machines can be optimized since it is similar to required space (resource-restricted scheduling). It can be used in problem settings similar to job-shop-scheduling problems (Gar76), where the number of machines has to be minimized and where the time is not relevant (see e.g. (GJ77)).
- Concurrent threads, independent of a programming language. The applicability of the algorithm depends on the specific programming language, in any case it can be expected that it is useful for independent threads.

This section is based on (SSD19a).

6.2.1 Abstract Model of Processes and Space

The assumptions underlying the abstraction is that processes use a common memory for their local data structures, but they cannot see each others areas in memory. The processes may independently start or stop or pause at certain time points. We also assume that synchronization and communication may occur at certain time points as interaction between processes.

Every process is abstractly modeled by its trace of space usage, given as a list of integers. In addition we later add constraints expressing simultaneous occurrences of time points of different processes as well as start- and end-points of processes.

In the following we write $tail(l)$ for the tail of list l and $[f(x) \mid x \in l]$ for a list l denotes the list of $f(x)$ in the same sequence as that of l, i.e. it is a list comprehension.

In the following we abstract a CHF*-process by a list of nonnegative integers. This is a reduction of the complex construct process to a trace of space values (i.e. the space consumption of a process), however for simplicity we call this list a *CHF*-process* or *process* in the rest of this section. A (parallel) interleaving is constructed such that from one state to the next one, each CHF*-process proceeds by at most one step and at least one CHF*-process proceeds. We formalize this in the following definition:

Definition 6.4 (CHF*-Process)
A *process* is a nonempty and finite list of nonnegative integers.
For $n > 0$ and n CHF*-processes $P_1, \ldots, P_n$, we write $len(p_i)$ for the length of p_i and $p_{i,j}$ for the elements, where $j \in \{1, \ldots, len(p_i)\}$.

An interleaving is defined as follows:

Definition 6.5 (Interleaving)
An *interleaving* of CHF*-processes $P_1, \ldots, P_n$ is a list $[q_1, \ldots, q_h]$ of n-tuples q_j constructed using the following nondeterministic algorithm:

1. Let q be the empty list.
2. If all CHF*-processes $P_1, \ldots, P_n$ are empty, then return q.
3. Set $q := q +\!+ [(p_{1,1}, \ldots, p_{n,1})]$, i.e. the tuple of all first elements is added at the end of q.
   Let $(b_1, \ldots, b_n)$ be a nondeterministically chosen tuple of Booleans, such that there is at least one k such that b_k is `True` and P_k not empty.
   For all $i = 1, \ldots, n$: Set $P_i = tail(P_i)$ if b_i and p_i is not empty, otherwise do not change P_i.
   Continue with step 2.

We also define certain terms and notations concerning processes and interleavings:

Definition 6.6 (Space-Usage, Required Space, Peaks and Valleys)
Let $P_1, \ldots, P_n$ be CHF*-processes.

1. The *space usage* $sps_I(S)$ of an interleaving S of $P_1, \ldots, P_n$ is the maximum of the sums of the elements in the tuples in S:
$$sps_I(S) = \max\left\{\sum_{i=1}^{n} a_i \mid (a_1, \ldots, a_n) \in S\right\}$$
2. The *required space* $spmin_I(P_1, \ldots, P_n)$ for n processes $P_1, \ldots, P_n$ is the minimum of the space usages of all interleavings of $P_1, \ldots, P_n$:
$$spmin_I(P_1, \ldots, P_n) := \min\{sps_I(S) \mid S \text{ is an interleaving of } P_1, \ldots, P_n\}$$
3. A *peak* of P_i is a maximal element of P_i and a *valley* is a smallest element of P_i.
4. A *local peak* of P_i is a maximal element in P_i which is not smaller than its neighbors. A *local valley* of P_i is a minimal element in P_i which is not greater than its neighbors.

To illustrate the definitions of CHF*-processes and interleavings, we give an example where a runtime-optimal scheduling is not space-optimal.

Example 6.1 (CHF*-Processes and Interleaving)
For two CHF*-processes $[1,7,3],[2,10,4]$ the *spmin$_I$*-value is 11, by first running the second one and then running the first. I.e. such a space-optimal interleaving is $[(1,2),(1,10),(1,4),(7,4),(3,4)]$.
The interleaving that results from a scheduling and tries to parallelize as much as possible, is $[(1,2),(7,10),(3,4)]$, with *sps*-value 17 and hence not space-optimal.

Programs with independent processes can be implemented in CHF* as follows:

$$u \overset{\text{main}}{\Longleftarrow} \ldots \mid \ldots \mid x_1 \Leftarrow e_1 \mid x_2 \Leftarrow e_2 \mid \ldots \mid x_n \Leftarrow e_n$$

If the threads $x \Leftarrow e_1$ to $x \Leftarrow e_n$ are independent from each other, then we can measure them separately and apply the notations above.

6.2.2 Standard Form of Processes

In this section we show, that a reduction by an iterated application of five patterns to processes, do not change the *spmin$_I$*-value and therefore simplify the search for a space-optimal schedule.

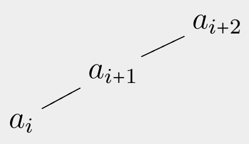
First we define some terms and notations:

Definition 6.7 (Pattern)
A pattern *matches a process* $[a_1,\ldots,a_k]$ at index i, if for index i the conditions of the pattern are satisfied.
A condition of a pattern is an equation (e.g. $a_i = a_{i+1}$), inequality or sequence of equations and inequalities (e.g. $a_i \le a_{i+1} \le a_{i+2}$).

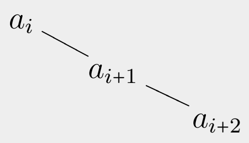
The straightforward patterns are defined as follows:

Definition 6.8 (Patterns M_0, M_1 and M_2)
The trivial pattern M_0 is $a_i = a_{i+1}$.

There are two further nontrivial patterns:
The first pattern M_1 is $a_i \le a_{i+1} \le a_{i+2}$, illustrated as follows:



The second pattern M_2 is $a_i \ge a_{i+1} \ge a_{i+2}$, illustrated as follows:



A single pattern application is as follows: If the patterns M_0, M_1 or M_2 matches a process for some index i, then a_{i+1} is removed.

We now show that M_0 does not affect the space measurement:

Lemma 6.1 (M_0 preserves *spmin*$_I$)
Let $P_1, \dots, P_n$ be n processes and let $P'_1, \dots, P'_n$ be the processes after removal of subsequent equal entries, i.e. using M_0. Then $\mathit{spmin}_I(P_1, \dots, P_n) = \mathit{spmin}_I(P'_1, \dots, P'_n)$.

Proof
This is obvious by rearranging the schedules, leading to different interleavings, which have the same *spmin*$_I$-value.

Also the patterns M_1 and M_2 do not change the *spmin*$_I$-value:

Lemma 6.2 (M_1 and M_2 preserve *spmin*$_I$)
Let $P_1, \dots, P_n$ be n processes. Let $P'_1, \dots, P'_n$ be the processes after several application of the pattern-reduction process using M_1 and M_2.
Then $\mathit{spmin}_I(P_1, \dots, P_n) = \mathit{spmin}_I(P'_1, \dots, P'_n)$.

Proof
It is sufficient to assume that exactly one change due to a pattern match is performed. It is also sufficient to assume that the pattern is M_1 and that it applies in P_1. We can also look only at a subpart of an interleaving to have easier to grasp indices. For argumentation purposes, we choose the correspondence between the interleaving $(P_1, P_2, \dots, P_n)$ and $(P'_1, P_2, \dots, P_n)$ as follows.

Let $[p_{1,1}, p_{1,2}, p_{1,3}]$ with $p_{1,1} \leq p_{1,2} \leq p_{1,3}$ be the subprocess of P_1 that is replaced by $[p_{1,1}, p_{1,3}]$. Consider the part

$$(p_{1,1}, \dots, p_{n,1}) : [(p_{1,2}, p_{2,2}, \dots, p_{n,2}) \mid (p_{2,2}, \dots, p_{n,2}) \in B]{+}{+}[(p_{1,3}, \dots, p_{n,3})]$$

of the interleaving, where B is a sequence of $n-1$-tuples. Then the modified interleaving for $(P'_1, P_2, \dots, P_n)$ can be defined as:

$$(p_{1,1}, \dots, p_{n,1}) : [(p_{1,1}, p_{2,2}, \dots, p_{n,2}) \mid (p_{2,2}, \dots, p_{n,2}) \in B]{+}{+}[(p_{1,3}, \dots, p_{n,3})]$$

For every interleaving of $(P_1, P_2, \dots, P_n)$ exists an interleaving of $(P'_1, P_2, \dots, P_n)$ with a *sps*$_I$ that is smaller or equal. Since *spmin*$_I$ is defined as a minimum, we obtain $\mathit{spmin}_I(P_1, P_2, \dots, P_n) \geq \mathit{spmin}_I(P'_1, P_2, \dots, P_n)$.

For the other direction, consider the part $[(p_{1,1}, \dots, p_{n,1}), (p_{1,3}, p_{2,2}, \dots, p_{n,2})]$ of an interleaving of the processes $P'_1, P_2, \dots, P_n$. Then *spmin*$_I$ of the part

$$[(p_{1,1}, \dots, p_{n,1}), (p_{1,2}, p_{2,2}, \dots, p_{n,2}), (p_{1,3}, p_{2,2}, \dots, p_{n,2})]$$

of the interleaving of $P_1, \dots, P_n$ is the same as before, thus the following inequality holds: $\mathit{spmin}_I(P_1, \dots, P_n) \leq \mathit{spmin}_I(P'_1, P_2, \dots, P_n)$

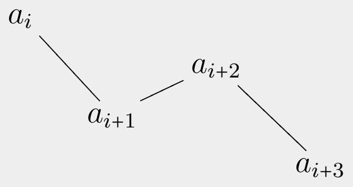
The two inequations imply $\mathit{spmin}_I(P_1, \dots, P_n) = \mathit{spmin}_I(P'_1, P_2, \dots, P_n)$.

If the patterns M_0, M_1 and M_2 are applied exhaustively, then this leads to a special form of processes:

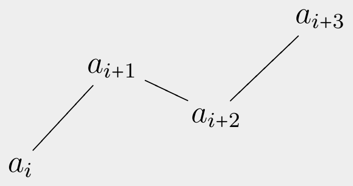
Definition 6.9 (Zig-Zag-Process)
If in a process P every strict increase is followed by a strict decrease and every strict decrease is followed by a strict increase, then the process P is called a *zig-zag process.*

Now we show that there are more complex patterns that can also be used to reduce the processes before computing *spmin*$_I$. The following patterns M_3 and M_4 are like stepping downstairs and upstairs, respectively.

Definition 6.10 (Patterns M_3 and M_4)
The patterns M_3 and M_4 are defined as follows:
M_3 consists of the values $a_i, a_{i+1}, a_{i+2}, a_{i+3}$ with $a_i > a_{i+1}$, $a_{i+1} < a_{i+2}$, $a_{i+2} > a_{i+3}$ and $a_i \geq a_{i+2}, a_{i+1} \geq a_{i+3}$.



M_4 consists of the values $a_i, a_{i+1}, a_{i+2}, a_{i+3}$ with $a_i < a_{i+1}$, $a_{i+1} > a_{i+2}$, $a_{i+2} < a_{i+3}$ and $a_i \leq a_{i+2}, a_{i+1} \leq a_{i+3}$.



If M_3 or M_4 matches for some i, then eliminate a_{i+1} and a_{i+2}.

We show that the complex patterns can be used to restrict the search for an optimum to special processes:

Lemma 6.3 (M_3 and M_4 preserve *spmin*$_I$)
Let $P_1, \ldots, P_n$ be n processes. Let $P'_1, \ldots, P'_n$ be the processes after several application of the pattern-reduction process using M_3 and M_4.
Then *spmin*$_I(P_1, \ldots, P_n) =$ *spmin*$_I(P'_1, \ldots, P'_n)$.

Proof
It is sufficient to assume that exactly one change due to a pattern match is performed. It is sufficient to assume that the pattern is M_3 and that it applies in P_1. We can also look only at a subpart of an interleaving to have easier to grasp indices. For argumentation purposes, we choose the correspondence between the interleaving $(P_1, P_2, \ldots, P_n)$ and $(P'_1, P_2, \ldots, P_n)$ as follows.

Let $[p_{1,1}, p_{1,2}, p_{1,3}, p_{1,4}]$ with $p_{1,1} > p_{1,2}$, $p_{1,2} < p_{1,3}$, $p_{1,3} > p_{1,4}$, $p_{1,1} \geq p_{1,3}$ and $p_{1,2} \geq p_{1,4}$ be the subprocess of P_1 that is replaced by $[p_{1,1}, p_{1,4}]$. Consider the following part of the interleaving, where B_2, B_3 are sequences of $n-1$-tuples:

$$(p_{1,1}, \ldots, p_{n,1}) : [(p_{1,2}, p_{2,2}, \ldots, p_{n,2}) \mid (p_{2,2} \ldots, p_{n,2}) \in B_2]$$
$$++[(p_{1,3}, p_{2,3}, \ldots, p_{n,3}) \mid (p_{2,3} \ldots, p_{n,3}) \in B_3]++[(p_{1,4}, \ldots, p_{n,4})]$$

The modified interleaving for $(P'_1, P_2, \dots, P_n)$ can be defined as

$$[(p_{1,1}, \dots, p_{n,1})]{+}{+}[(p_{1,4}, q_{2,4}, \dots, q_{n,4}) \mid (q_{2,4}, \dots, q_{n,4}) \in B_2{+}{+}B_3{+}{+}[p_{2,4}, \dots, p_{n,4}]]$$

and since for every interleaving of $(P_1, P_2, \dots, P_n)$ we obtain an interleaving of $(P'_1, P_2, \dots, P_n)$ with a *sps*$_I$ that is smaller or equal and since *spmin*$_I$ is defined as a minimum, we obtain *spmin*$_I(P_1, P_2, \dots, P_n) \geq$ *spmin*$_I(P'_1, P_2, \dots, P_n)$.

For the other direction we consider the part $[(p_{1,1}, \dots, p_{n,1}), (p_{1,4}, \dots, p_{n,4})]$ of an interleaving of $P'_1, P_2, \dots, P_n$. Then *spmin*$_I$ of the part

$$[(p_{1,1}, p_{2,1}, \dots, p_{n,1}), (p_{1,2}, p_{2,1} \dots, p_{n,1}), (p_{1,3}, p_{2,1}, \dots, p_{n,1}), (p_{1,4}, \dots, p_{n,4})]$$

of the interleaving of $P_1, \dots, P_n$ is the same as before, thus the following inequality holds: *spmin*$_I(P_1, \dots, P_n) \leq$ *spmin*$_I(P'_1, P_2, \dots, P_n)$

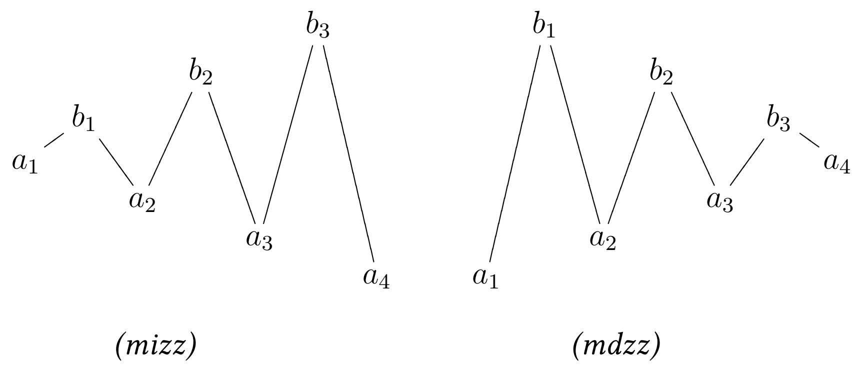
The two inequations imply *spmin*$_I(P_1, \dots, P_n) =$ *spmin*$_I(P'_1, P_2, \dots, P_n)$.

We define three zig-zag-variants for processes, that ease the search for the required space.

Definition 6.11 (Zig-Zag-Variants mizz, mdzz and midzz)

1. A process $[a_1, b_1, a_2, b_2, \dots, a_n]$ (or $[b_0, a_1, b_1, a_2, \dots, a_n]$ or $[b_0, a_1, b_1, a_2, b_2, \dots, a_n, b_n]$ or $[a_1, b_1, a_2, b_2, \dots, a_n, b_n]$, resp.) is a *monotonic increasing zig-zag (mizz)*, iff $a_i < b_j$ for all i and j, $a_1, a_2, \dots, a_n$ is strictly monotonic decreasing and $b_1, b_2, \dots, b_{n-1}$ (and $b_0, b_1, b_2, \dots, b_{n-1}$ and $b_0, b_1, b_2, \dots, b_{n-1}, b_n$, resp.) is strictly monotonic increasing.
2. A process $[a_1, b_1, \dots, a_n]$ is a *monotonic decreasing zig-zag (mdzz)*, iff $a_i < b_j$ holds for all i and j, $a_1, a_2, \dots a_n$ is strictly monotonic increasing and $b_1, b_2, \dots, b_{n-1}$ (or $b_0, b_1, b_2, \dots, b_{n-1}$, resp.) is strictly monotonic decreasing.
3. A process is a *midzz*, if it is a mizz followed by a mdzz. More rigorously, there are essentially two cases, where we omit the cases with end-peaks and/or start-peaks:
   - The mizz $[a_1, b_1, a_2, b_2, \dots, a_n]$ and the mdzz $[a'_1, b'_1, \dots, a'_n]$, where $a_n = a'_1$ are combined to $[a_1, b_1, a_2, b_2, \dots, a_n, b'_1, \dots, a'_n]$.
   - The mizz $[a_1, b_1, a_2, b_2, \dots, a_n, b_n]$ and the mdzz $[b'_0, a'_1, b'_1, \dots, a'_n]$, where $b_n = b'_0$ are combined to $[a_1, b_1, a_2, b_2, \dots, a_n, b_n, a'_1, b'_1, \dots, a'_n]$.

Typical graphical representations of mizz- and mdzz-sequences are:



(mizz) *(mdzz)*

If the goal is to compute the required space, then there are several reduction operations on processes that ease the computation and help us to concentrate on the hard case. First we show that one-element processes can be excluded.

Proposition 6.1 (Exclusion of One-Element-Processes)
If $P_1 = [a_1]$ and $P_2, \ldots, P_n$ are processes then the following holds:

$$\mathit{spmin}_I(P_1, \ldots, P_n) = a_1 + \mathit{spmin}_I(P_2, \ldots, P_n)$$

Proof
a_1 is the first element of every tuple in any interleaving of $P_1, \ldots, P_n$, hence the claim is valid.

Processes with start- or end-peaks can be reduced by omitting elements.

Proposition 6.2 (Omission of Start- and End-Peaks)
Let $P_i = [p_{i,1}, \ldots, p_{i,n_i}]$ for $i \in \{1, \ldots, n\}$ be processes. If $p_{1,1}$ is a start-peak of P_1, then let $P_1' = [p_{1,2}, \ldots, p_{1,n_1}]$.
Then $\mathit{spmin}_I(P_1, \ldots, P_n) = \max(\sum_i p_{i,1}, \mathit{spmin}_I(P_1', P_2, \ldots, P_n))$.
The same holds symmetrically if P_1 ends with a local peak.

Proof
Consider the following interleaving for $P_1, \ldots, P_n$ and some h:

$$q = [(p_{1,1}, q_{1,2}, \ldots, q_{1,n}), \ldots, (p_{1,1}, q_{h,2}, \ldots, q_{h,n})] ++ [(p_{1,2}, q_{h+1,2}, \ldots, q_{h+1,n})] ++ R$$

If $h \neq 1$, this can be changed to

$$[(p_{1,1}, q_{1,2}, \ldots, q_{1,n}), (p_{1,2}, q_{2,2}, \ldots, q_{2,n}), \ldots, (p_{1,2}, q_{h,2}, \ldots, q_{h,n})] \\ ++[(p_{1,2}, q_{h+1,2}, \ldots, q_{h+1,n})] ++ R$$

without increasing the necessary space. Hence the following inequality holds:

$$\mathit{spmin}_I(P_1, \ldots, P_n) \geq \max\left(\sum_i p_{i,1}, \mathit{spmin}_I(P_1', P_2, \ldots, P_n)\right)$$

On the other hand, if we have a space-optimal schedule of $P_1', P_2, \ldots, P_n$, then we can extend this by starting with $(p_{1,1}, \ldots, p_{n,1})$ and obtain the following inequality: $\mathit{spmin}_I(P_1, \ldots, P_n) \leq \max(\sum_i p_{i,1}, \mathit{spmin}_I(P_1', P_2, \ldots, P_n))$.

Hence $\mathit{spmin}_I(P_1, \ldots, P_n) = \max(\sum_i p_{i,1}, \mathit{spmin}_I(P_1', P_2, \ldots, P_n))$.

Since start- and end-peaks were removed and also singleton processes are not need to be considered by the main calculation, the following holds:

Lemma 6.4 (Minimal Length of Processes)
We can assume that processes $P_1, \ldots, P_n$ are all of length at least 3 for computing the optimal space.

Proof
Proposition 6.1 requires a length of at least 2 and Proposition 6.2 requires that there is neither a start- nor an end-peak. Hence we obtain the the claim.

The patterns cannot remove start- or end-peaks and preserve start- and end-valleys, hence Proposition 6.2 needs to be applied before the main calculation.

Proposition 6.3 (Patterns $M_0, \dots, M_4$ preserve Start- and End-Valleys)
Let P be a process that starts and ends with local valleys. Then the application of the patterns $M_0, \dots, M_4$ with subsequent reduction always produces a process that also starts and ends with local valleys.

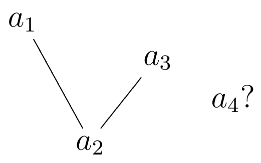
Proof
The reduction either removes according to pattern M_0 or it removes inner entries of the lists.

Now we show that through the exhaustive use of the five patterns $M_0, \dots, M_4$ for reductions a midzz is calculated, if start- and end-peaks are also eliminated.

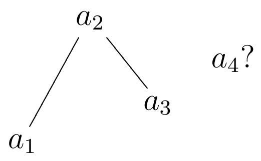
Proposition 6.4 (Generation of midzz)
A process such that none of the patterns M_0, M_1, M_2, M_3, M_4 matches and which does neither start nor end with a local peak is a midzz.

Proof
We consider all four different cases how small sequences may proceed, if no pattern applies.

1. Case $a_1 > a_2$, $a_2 < a_3$ and $a_3 < a_1$. Then $a_4 < a_3$. The relation $a_4 \leq a_2$ is not possible, since then pattern M_3 matches. Hence $a_3 > a_4 > a_2$. Then a_1, a_2, a_3, a_4 is a tail of a mdzz.
   The case $a_1 > a_2$, $a_2 < a_3$ and $a_3 = a_1$ leads to the same relations $a_3 > a_4 > a_2$. Then a_2, a_3, a_4 is a tail of a mdzz.

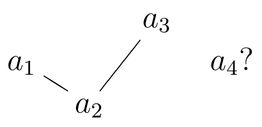


2. Case $a_1 < a_2$, $a_2 > a_3$ and $a_3 > a_1$. Then $a_4 > a_3$. The relation $a_4 \geq a_2$ is not possible, since then pattern M_4 matches. Hence $a_3 < a_4 < a_2$. Then a_1, a_2, a_3, a_4 is a mdzz.
   The case $a_1 < a_2$, $a_2 > a_3$ and $a_3 = a_1$ leads to the same relations $a_3 < a_4 < a_2$. Then using case 1 for the the next element a_5, the sequence a_3, a_4, a_5 is a tail of a mdzz.

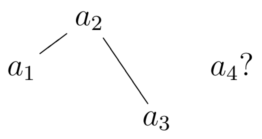


3. Case $a_1 > a_2$, $a_2 < a_3$ and $a_3 > a_1$. Then $a_3 > a_4$ and there are three cases:
   - If $a_4 = a_2$ then the sequence starting from a_3 is a mdzz.

- If $a_4 > a_2$ then case 2 is applicable and the sequence starting from a_2 is a mdzz.
- If $a_4 < a_2$ then the sequence starting with a_1 proceeds as mizz. It may later turn into a mdzz.



4. Case $a_1 < a_2$, $a_2 > a_3$ and $a_3 < a_1$. Then $a_3 < a_4$ and there are three cases:
   - If $a_4 = a_2$ then the sequence starting from a_3 is a mdzz.
   - If $a_4 < a_2$ then case 1 is applicable and the sequence starting from a_2 is a mdzz.
   - If $a_4 > a_2$ then the sequence starting with a_1 proceeds as mizz. It may later turn into a mdzz.



Now we put the parts together and conclude that the sequence must be a midzz.

Note that the definition of midzz permits the simplified case that the process is a mizz or mdzz. Finally, a standard form of processes can be defined:

Definition 6.12 (Standardized Process)
A process is called *standardized* if it is a midzz of length at least 3 and does neither start nor end with a local peak.

For the calculation of the required space, it is important, that there are not two global valleys and two global peaks at the same time.

Lemma 6.5 (Number of Global Valleys and Global Peaks)
Let P be a midzz-process, where none of the patterns M_0, M_1, M_2, M_3 and M_4 applies, has a length of at least 3 and does neither start nor end with a local peak. Then a midzz-process has one or two global peaks, it has one or two global valleys, but not two global peaks and two global valleys at the same time.

Proof
The considerations and cases in the proof of Proposition 6.4 already exhibit the possible cases.

Since the patterns M_3 and M_4 do not apply, there cannot be three global peaks and also there cannot be three global valleys. If there are two global peaks and two global valleys, then the picture is as follows:

$$\begin{tikzcd}
a_1 \arrow[dr, no head] & & a_3 \arrow[dl, no head] \arrow[dr, no head] & \\
& a_2 & & a_4
\end{tikzcd}$$

In this case we can apply pattern M_3, which is forbidden by the assumptions.

The case where a_1 is a global valley is similar.

Hence, a standardized process in midzz-form has three different possibilities for the global peaks and valleys:

1. A unique global peak and a unique global valley.
2. A unique global peak and two global valleys.
3. Two global peaks and a unique global valley.

In summary, we can use standardized processes for the calculation of the required space, where the $spmin_I$-value is not affected by the standardization. The possibilities of global valleys and peaks above also ease the calculation.

6.2.3 Required Space of Many Independent Processes

In this section we give an algorithm that calculates the required space of many independent processes. First we give an algorithm that implements the standardization of Section 6.2.2.

Algorithm 6.1 (Standardization of N Processes)
For an input of N processes $P_1, \ldots, P_N$:

1. For every process P_i in turn scan P_i by iterating j from 0 as follows:
   If the patterns $M_0, \ldots, M_4$ apply at index j then reduce accordingly and restart the scan at position $j - 3$, otherwise go on with index $j + 1$.
2. Let K_0 be the sum of all first elements of $P_1, \ldots, P_N$.
   Let $P'_1, \ldots, P'_N$ be obtained from $P_1, \ldots, P_N$ by removing all start-peaks only from processes of length at least 2.
3. Let K_ω be the sum of all last elements of $P'_1, \ldots, P'_N$.
   Let $P''_1, \ldots, P''_N$ be obtained from $P'_1, \ldots, P'_N$ by removing all end-peaks only from processes of length at least 2.
4. Let A be the sum of all elements of one-element processes and let $P'''_1, \ldots, P'''_{N'}$ be $P''_1, \ldots, P''_N$ after removing all one-element processes.
5. If M''' is $spmin_I(P'''_1, \ldots, P'''_{N'})$, then $spmin_I(P_1, \ldots, P_N)$ is computed as $\max(M''' + A, K_0, K_\omega)$.

We show the correctness of the above algorithm and analyze the runtime:

Theorem 6.1 (Correctness and Complexity of Standardization of N Processes)
Algorithm 6.1 for standardization reduces the computation of $spmin_I$ for N processes $P_1, \ldots, P_N$ of size n to the computation of $spmin_I$ for standardized processes in runtime $O(n)$.

Proof
Algorithm 6.1 is correct by Lemma 6.1, Lemma 6.2, Lemma 6.3, Proposition 6.1, Proposition 6.2 and Proposition 6.4.

The required number of steps for pattern application is $O(n)$: Every successful application of a pattern strictly reduces the number of elements. The maximum number of steps back is 3, hence at most $4n$ total steps are necessary. Stepping back for 3 is correct, since a change at index k cannot affect pattern application for indices less than $k - 3$. The overall complexity is $O(n)$ since scans are sufficient to perform all the required steps and computations in Algorithm 6.1.

Since a zig-zag-process is symmetric, the calculation of the required space can be split into two parts: From the left to the middle and from the right to the middle using symmetry. Hence the following algorithm only works on the left half. Note that in the case, where a midzz is a mizz, the algorithm is also applicable and in the case of a mdzz the symmetric variant of the algorithm applies.

Algorithm 6.2 (Left-Scan of N processes)
We describe an algorithm for standardized processes which performs a left-scan until a global valley is reached and returns the required space for the left part.
The following index $I_{i,ends}$ in process P_i for $i \in \{1, \dots, N\}$ is fixed: It is the index in P_i of the global valley, if it is unique and of the rightmost global valley if there are two global valleys.

1. Build up a search tree T that contains $((p_{i,2} - p_{i,1}), i)$ for each $P_i = [p_{i,1}, \dots, p_{i,n_i}]$, where the first component is the search key.
2. Set $S = M = \sum_i p_{i,1}$. Also for each process P_i there are indices I_i indicating the current valley positions of the process, initially set $I_i = 1$ for each process.
3. If T is empty then return M and terminate.
4. Remove the minimal element $V = (d, i)$ from T.
   If $I_i + 2 \leq I_{i,ends}$, then set $M = \max(M, S + d)$, $S = S + (p_{i,3} - p_{i,1})$, insert $(p_{i,4} - p_{i,3}, i)$ into T (only if P_i contains at least 4 elements), set $I_i = I_i + 2$ and remove the first two elements from P_i. Note that P_i is not considered anymore in the future if $I_i + 2 > I_{i,ends}$ or if there is no further peak in P_i after I_i.
   Goto (3).

The algorithm for the right-scan is the symmetric version and also yields the required space for the right part, that only scans to the rightmost valley for every process.

We show the correctness and analyze the runtime for the left- and right-scan.

Theorem 6.2 (Correctness and Complexity of Left- and Right-Scan)
Algorithm 6.2 calculates the required space *spmin*$_I$ for N standardized processes until a global valley is reached. The runtime is $O((N + n) \log N)$, where n is the total size of the processes.
The same holds for the right-scan symmetrically, where the scan only runs to the rightmost global valley.

Proof
Consider a state $(i_1, \ldots, i_n)$ during the construction of a space-optimal interleaving using space M, where every i_j is not after the index of the smallest valley, which means $i_j \leq I_{j,ends}$.

An invariant of the state is that $p_{i_1} + \ldots + p_{i_n} \leq M$ holds. We also assume as an invariant that the current state belongs to an optimal interleaving.

If some i_j is the position of a local peak, then the optimal interleaving can be changed to $i_j + 1$ such that the next tuple is $(i_1, \ldots, i_j + 1, \ldots, i_n)$. Repeating this argument, we can assume that $(i_1, \ldots, i_n)$ contains only indices of local valleys. Now consider the set S of positions j in the tuple, such that $i_j < I_{j,ends}$. For at least one such index the optimal interleaving must proceed.

For the indices in S, the next index will be a local peak, so the best way is to look for the smallest peak p_{i_j+1} for $j \in S$. If the sum of the spaces exceeds M then we have a contradiction, since the interleaving must proceed somewhere. Hence M is at least $\min\{p_{i_j+1} + \sum_{h \neq j} p_{i_h} \mid j = 1, \ldots, n\}$. This argument also holds, if the indices i_j for $j \notin S$ are beyond $I_{j,ends}$, since the valley at $I_{j,ends}$ is smaller.

For a better efficiency the algorithm calculates these sums implicitly by keeping track of the sum of the current valleys, i.e. $\sum_h p_{i_h}$. Then it uses a search tree containing the space differences between the corresponding local valley and the next peak to step forward, i.e. to calculate p_{i_j+1}. The search tree can be initially constructed in $O(N \log N)$. Since the search tree contains at most N elements during the whole calculation, we need $O(n \log N)$ steps for all lookups and insertions. Hence the overall runtime of the left-scan is $O(N \log N + n \log N)$.

For the right-to-left scan the same arguments hold, symmetrically where by slight asymmetry, we only scan to the rightmost global valley for every process.

Using the algorithms above we now can construct an algorithm, that calculates the required space for N processes:

Algorithm 6.3 (SpOptN Computation of *spmin*$_I$ for N processes)
1. Let M_{start} be the sum of all start elements and M_{end} be the sum of all end elements of the given processes $P_1, \ldots, P_N$.
   Also let M_{one} be the sum of all elements of one-element-processes.
2. Transform the set of processes into standard form using Algorithm 6.1.
3. Compute M_{left} using the left-scan and M_{right} using the right-scan (see Algorithm 6.2).
4. Return the maximum of $(M_{left} + M_{one})$, $(M_{right} + M_{one})$, M_{start} and M_{end}.

We also show the correctness of the complete algorithm SpOptN and analyze the runtime.

Theorem 6.3 (Correctness and Complexity of SpOptN)
Algorithm 6.3 calculates the required space *spmin*$_I$ for N processes in runtime $O((N + n) \log N)$, where n is the total size of the processes.

Proof
The standard forms of the processes is achieved by Algorithm 6.1, hence the correctness and runtime $O(n)$ of this part is shown by Theorem 6.1.

The values M_{left} and M_{right} are calculated using Algorithm 6.2 and its symmetric variant, hence the correctness and runtime $O((N+n)\log N)$ for this part is shown by Theorem 6.2.

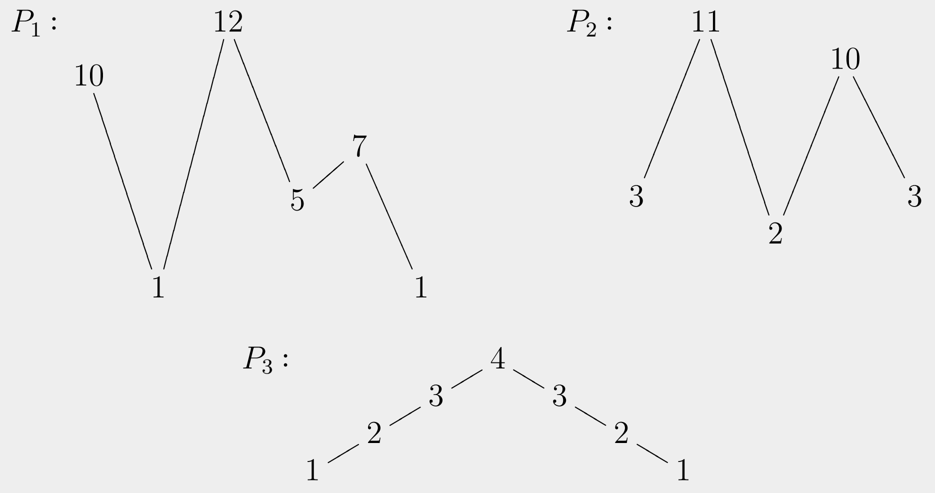
The only missing argument is that we can combine M_{left} and M_{right}. For processes that have a unique global minimal valley the combination is trivial. For the case of processes that have global minimal valleys, we glue together the left interleaving with the reversed right interleaving. This is an interleaving and it can be performed in space at most the maximum of M_{left} and M_{right}.

Note that the bit-size of the integers of the space-sizes is not relevant, since we only use addition, subtraction and maximum-operations on these numbers.

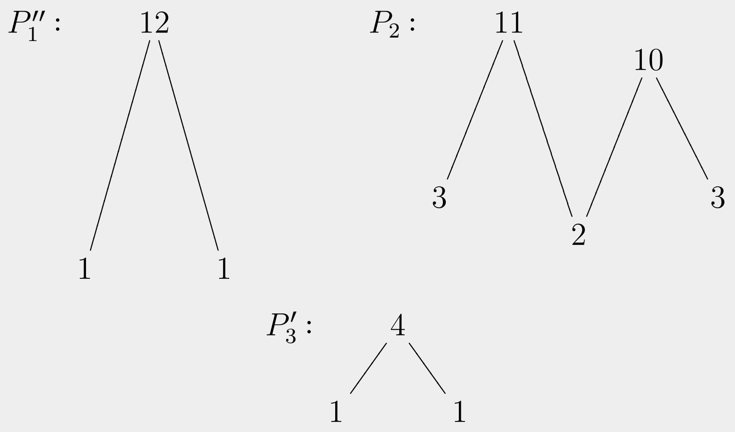
Concluding, the algorithm computes *spmin*$_I$ for the input processes with runtime $O((N+n)\log N)$.

We give an example for the algorithm SpOptN.

Example 6.2 (Computation using SpOptN)
This example illustrates the computation of SpOptN (see Algorithm 6.3) as follows. The optimization using search trees is not considered in this example, since the unoptimized variant is easier to grasp.
Let $P_1 = [10, 1, 12, 5, 7, 1]$, $P_2 = [3, 11, 2, 10, 3]$ and $P_3 = [1, 2, 3, 4, 3, 2, 1]$. The processes can be sketched as follows:



Then we first can reduce the processes as follows: P_3 can be reduced by patterns M_1 and M_2 to $P_3' = [1, 4, 1]$. P_2 is already a zig-zag process, therefore no pattern applies. P_1 starts with a local peak, hence we keep in mind 14 as the sum of the first elements and replace P_1 by $P_1' = [1, 12, 5, 7, 1]$. The next step is to apply the pattern M_3, which reduce it to $P_1'' = [1, 12, 1]$. Thus the new problem is $P_1'' = [1, 12, 1]$, $P_2 = [3, 11, 2, 10, 3]$ and $P_3' = [1, 4, 1]$ as sketched in the following:



A short try shows that 15 is the optimum. However, we want to demonstrate the algorithm:
The left scan starts with $M = 5$. The peak in P_3' then enforces $M = 8$ and P_3' is not considered anymore, since the left scan reached the final position in P_3, i.e. the right-most global valley. The peak in P_2' then enforces $M = 13$ and also P_2 is not considered anymore, since the final position is reached. Finally the peak in P_1'' enforces $M = 15$ and the left scan terminates.
The right scan starts with $M = 5$. Then the peak in P_3' enforces $M = 8$, after this the peak in P_2' enforces $M = 12$ and finally the peak in P_1'' enforces $M = 15$.
Hence in summary, also taking the local peak at the beginning of P_1 into account, the result is 15.

6.2.4 Processes with Synchronizations

The model of independent processes can be extended to timing and synchronization restrictions. For example, in CHF* writing into a filled MVar requires the process to wait until the MVar is empty. There are also race-conditions, for example if several processes try to write into an empty MVar or several processes try to read the same MVar. These cases are captured by the constraints below, where the race conditions can be modeled by disjunctions.

Definition 6.13 (Basic Synchronization Restrictions)
There may be various forms of synchronization restrictions. We will only use the following forms of fundamental restrictions:

1. *simul*(P_1, P_2, i_1, i_2): For processes P_1 and P_2 the respective actions at indices i_1 and i_2 must happen simultaneously.
2. *starts*(P_1, P_2, i): Process P_1 starts at index i of process P_2.
3. *ends*(P_1, P_2, i): Process P_1 ends at index i of process P_2.
4. *before*(P_1, P_2, i_1, i_2): For processes P_1 and P_2 the action at index i_1 of P_1 happens simultaneously or before the action at i_2 of P_2.

For a set R of restrictions only schedules are permitted that obey all restrictions. This set R is also called a set of *basic restrictions.*
We also permit Boolean formulas over such basic restrictions. In this case the permitted schedules must obey the complete formula.

Note that in CHF* these restrictions correspond to synchronization conditions of the start of a future or waiting for an MVar to be in the right state. The simultaneous condition is not necessary for single reduction steps in CHF*, but can be used for blocks of monadic commands.

We show that there is an algorithm for computing the optimal space and an optimal schedule that has an exponential complexity, where the exponent is $b \cdot N$ where b is the size of the Boolean formula and N is the number of processes.

Theorem 6.4 (Upper Complexity of Synchronization Restrictions)
Let there be N processes and a set B of Boolean restrictions where b is the size of B and n the size of the input. Then there is an algorithm to compute the optimal space and an optimal schedule of worst case asymptotic complexity of $O(poly(n) \cdot n^{O(b \cdot N)})$, where $poly$ is a polynomial.

Proof
The algorithm is simply a brute force method of trying all possibilities:

For every condition try all tuples of indices. The number of different tuples is at most n^N and for trying this for every basic restriction we get an upper bound of $n^{N \cdot b}$.

Now we have to check whether the time constraints are valid, i.e. there are no cycles, which can be done in polynomial time. Then we can split the problem into at most $b + 1$ intervals with interception of an index of a condition and apply for every interval the algorithm SpOptN (see Algorithm 6.3), which requires time sub-quadratic in n by Theorem 6.3.

Thus we get an asymptotic time complexity as claimed.

However for a fixed number of processors and a fixed size of Boolean restrictions, the time is obviously polynomial:

Corollary 6.1 (Polynomial Time for Fixed Number of Processors and Restrictions)
Let there be N processes and a set B of Boolean restrictions where b is size of B and n the size of the input. Assume that the number N of processes and the size of B is fixed. Then there is a polynomial algorithm to compute the optimal space and an optimal schedule.

In general, the synchronization restrictions yield NP-completeness:

Theorem 6.5 (NP-Completeness of General Synchronization Restrictions)
Using general synchronization restrictions, the problem of finding the required space is NP-hard and hence NP-complete.

Proof
We use the (perfect) partition problem, which is known to be NP-hard. An instance is a multi-set A of positive integers and the question is whether there is a partition of A into two sub-multi-sets A_1 and A_2 such that $\sum A_1 = \sum A_2$.

This can be encoded as the question for the minimal space for a scheduling:

Let $P_i = [0, a_i, 0, 0]$ for $A = \{a_1, \ldots, a_n\}$ and $P_0 = [0, 0, 0, 0]$, where the indices are $1, 2, 3, 4$. The condition is a conjunction of the following disjunctions: $(P_0, P_i, 2, 3) \vee (P_0, P_i, 3, 2)$. The optimal space is reached for a schedule, where the indices 1 and 4 are zero and where at index 2 and 3, there is a perfect partition of A.

Since SpOptN solves a special case, it is natural to search for an approximation algorithm that solves the space-optimization using general synchronization restrictions. In (MSV12) tree-shaped task graphs are used, where the task durations and number of processors are assumed to be known. They showed that the problem, whether there exists a schedule of a task graph, that has a makespan lower or equal to a given constant and also a peak memory consumption does not exceed a given constant, is NP-complete. Furthermore they showed that there exists no approximation algorithm with constant approximation factors for makespan and peak memory consumption.

6.2.5 Applications of Space-Optimizations using Synchronizations

Apart from space analyses in CHF* there are two further applications of finding space-optimal schedules using synchronizations.

Producer-Consumer Problem

We illustrate how an abstract version of the producer-consumer problem can be modeled using interleavings and synchronization restrictions. The idea is that the consumer process P_1 produces a list or stream that is consumed by the process P_2. The single elements are also modeled as processes.

Our model will be such that the optimal space model coincides with the intuition that the space usage of the intermediate list is minimal if there is an eager consumption of the produced list elements.

We represent the problem as follows. There are two processes P_1 and P_2, the producer and the consumer, which consist of n times the symbol 1. There are also n processes $Q_1, \ldots, Q_n$ that only consist of two elements: A 1 followed by a 0, where the processes represent the unconsumed parts of the exchanged list.

We represent the possible executions by synchronization restrictions:

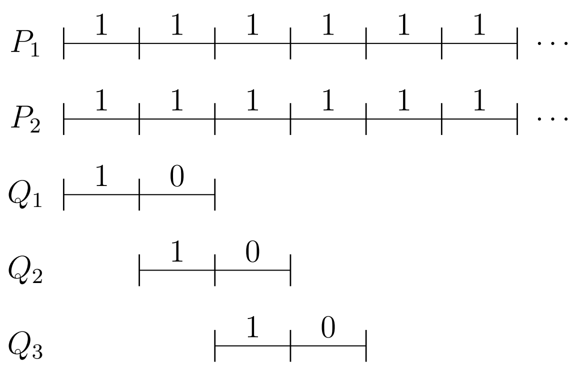
- Q_i is started by P_1 at time point i:

  starts(Q_i, P_1, i)
- Q_i is consumed by P_2 at a time point i or later:

  before$(P_2, Q_i, i, 2)$ for all i.
- Q_{i+1} ends later than Q_i for all i:

  before$(Q_i, Q_{i+1}, 2, 2)$ for all i.

The start of the space-optimal schedule is as follows and requires 3 units of space:



Variant of Job Shop Scheduling

A variant of job shop scheduling is the following:

Let there be n jobs (processes) that have to be performed on a number of identical machines. If the focus is on the question how many machines are sufficient for processing, then we can ignore the time and thus only specify the number of machines that are necessary for every single sub-job of any job. The necessary information is then the list of numbers (of machines) for every job. Note that also the number 0 is permitted. The trivial solution would be that all jobs run sequentially, in case the machine lists of every job are of the form $[0, k_2, \ldots, k_n, 0]$.

If there are in addition (special) time constraints, for example every job starts immediately with a nonzero number of machines, all jobs end with a nonzero number of machines and they terminate all at the same time, then our algorithm SpOptN (see Algorithm 6.3) can be applied in a nontrivial way and will compute the minimal total number of necessary machines.

In the case of further time constraints, Corollary 6.1 shows that in certain cases there are efficient algorithms and Theorem 6.5 shows that the problem, if there are general time constraints, is NP-complete.

Our approach and algorithm SpOptN (see Algorithm 6.3) is related to resource constrained project scheduling (ADN08) insofar as we are looking and optimizing the space resource of several given processes (projects respectively). The difference is that in job shop and project scheduling the primary objective is to minimize the overall required time, whereas our algorithm computes a minimal bound of a resource (here space) not taking the time into account.

6.3 Environment for Space Analyses

Since CHF*-programs often have synchronization points and the general case of synchronization restrictions is NP-complete (see Theorem 6.5) this motivates to develop an environment for space analyses in CHF*. Moreover such an environment helps to find examples, where a transformation increases the space consumption and also we can perform more complex analyses.

6.3.1 Abstract Machine CIOM1$_{sp,int}$

The abstract machine CIOM1 as defined in Section 2.4.4 implements a sequential standard reduction. We adapt the abstract machine, so that the space consumption is compatible to CHF* using *spmin* (see Definition 6.3). The result is the abstract machine CIOM1$_{sp,int}$.

First of all we give a definition of a garbage collector as an additional transition rule for the abstract machine CIOM1, that has to be applied whenever possible to be compatible to the eager garbage collection approach of the measures *sps* and *spmin* (compare Definition 6.3):

Definition 6.14 (Additional CIOM1-Rule (GC))
(GC) $\langle \mathcal{H}, \{x_i = s_i\} \mid \mathcal{M}, \{y_j = t_j\} \mid \mathcal{T}\rangle \to \langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T}\rangle$
where $\mathcal{T} = \{(n_1, e_1, \mathcal{S}_1, \mathcal{I}_1), \dots, (n_m, e_m, \mathcal{S}_m, \mathcal{I}_m)\}$.

$\{x_i = s_i\}$ is the maximal set such that for all i and $k \in \{1, \dots, m\}$:
$x_i \notin FV(\mathcal{H}), x_i \notin FV(e_k), \#\text{app}(x_i) \notin \mathcal{S}_k, \#\text{seq}(x_i) \notin \mathcal{S}_k, \#\text{put}(x_i) \notin \mathcal{I}_k,$
$\#\text{bind}(x_i) \notin \mathcal{I}_k$ and if $x_i \in FV(alts)$ then $\#\text{case}(alts) \notin \mathcal{S}_k$

$\{y_j = t_j\}$ is the maximal set such that for all j and $k \in \{1, \dots, m\}$:
$y_j \notin FV(\mathcal{H}), y_j \notin FV(e_k), \#\text{app}(y_j) \notin \mathcal{S}_k, \#\text{seq}(y_j) \notin \mathcal{S}_k, \#\text{put}(y_j) \notin \mathcal{I}_k,$
$\#\text{bind}(y_j) \notin \mathcal{I}_k$ and if $y_j \in FV(alts)$ then $\#\text{case}(alts) \notin \mathcal{S}_k$

Since $\mathcal{H}$ only contains bindings, $\mathcal{M}$ only contains MVars and (GC) only removes bindings of both $\mathcal{H}$ and $\mathcal{M}$, this is a direct implementation of (gc) (compare Definition 6.2). The requirements for stack and IO-stack are needed, because the CIOM1 stores parts of the program on stack and IO-stack during execution.

The space measure *sps* (see Definition 6.3) abstracts over local peaks, since only the maximum of such values that do not allow a garbage collection are considered. This transfers to the space measure *spmin* that is defined as the standard reduction sequence with the minimal *sps*-value. Often an abstraction or constructor application can be directly garbage collected, this both applies to CHF* and CIOM1 (the example in Section 3.6.1 also is applicable here).

The measure `msize` of M1 (see Definition 3.16) is now generalized for CIOM1. The size of a thread is defined as follows:

Definition 6.15 (Size of CIOM1-Thread `ctmsize`)
The size of a CIOM1-thread $T = (x, s, \mathcal{S}, \mathcal{I})$ is defined as follows:

The size of stack $\texttt{ctmsize}_{\mathcal{S}}$ is the sum of the sizes of the stack-entries. #app(x) and #seq(x) are counted as 1, #upd(x) as 0 and #case$(alts)$ as follows:
For a `case`-alternative $(c\ y_1\ \dots\ y_n) \to t$ the size is defined as $1 + \texttt{size}(t)$ and the size of #case$(alts)$ is the sum over all of such sizes of the `case`-alternatives $alts$.

The size of IO-Stack $\texttt{ctmsize}_{\mathcal{I}}$ is the sum of the sizes of the stack-entries, where #take, #put(x) and #bind(x) are counted as 1.

Finally $\texttt{ctmsize}(T) := \texttt{size}(s) + \texttt{ctmsize}_{\mathcal{S}} + \texttt{ctmsize}_{\mathcal{I}}$.

Using the definition of `ctmsize`, we can define the size of a CIOM1-state:

Definition 6.16 (Size of CIOM1-State `cmsize`)
The size of a CIOM1-state $S = \langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T} \rangle$ is defined as follows:
Let $\mathcal{H} = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$, then the size of the heap $\texttt{cmsize}_{\mathcal{H}}$ is defined as $\sum_{i=1}^{n} \texttt{size}(e_i)$.
Since $\mathcal{M}$ can be seen as heap, we define $\texttt{cmsize}_{\mathcal{M}}$ analogous to $\texttt{cmsize}_{\mathcal{H}}$.
The size of threads $\texttt{cmsize}_{\mathcal{T}}$ is the sum of the thread-sizes `ctmsize` of all of the threads in $\mathcal{T}$.
Finally $\texttt{cmsize}(S) := \texttt{cmsize}_{\mathcal{H}} + \texttt{cmsize}_{\mathcal{M}} + \texttt{cmsize}_{\mathcal{T}}$.

LRPgc copies values directly to the end of variable-to-variable-chains, without placing it at any intermediate position. Therefore in Section 3.6.1 we had to take care of this, since the abstract machine M1 implicitly copied values to each intermediate position of a variable-to-variable chain. In contrast to LRPgc, CHF* copies such values to the intermediate positions and also the abstract machine CIOM1 performs this implicit step-by-step copies, hence with regard to this there is no extra work required to obtain compatibility between CHF* and CIOM1.

In contrast to LRPgc, CHF* has no rules like (seq-in) or (case-in), hence the values and constructor applications has to be copied to the certain positions and then are directly evaluated by (seq) or (case). This is the reason why $\mathrm{M1}_{sp}$ needed a special space measurement, that ignored an (Update) if it is directly followed by (Seq) or (Case) (see Definition 3.17), to gain space-compatibility. This is not needed for CHF*, hence the definition of the overall space measurement of evaluation sequences of the CIOM1 is as follows:

Definition 6.17 (CIOM1 Space Measures sps_M and $spmin_M$)
Let $Red = S_1 \to \cdots \to S_n$ be a terminating evaluation sequence of the machine CIOM1. Then the space measure sps_M is defined as follows:

$$sps_M(Red) := \max\{\texttt{cmsize}(S_i) \mid \text{(GC) is not applicable to } S_i\}$$

The space measure for a state of the CIOM1 is defined as follows. Note that the state $\langle \emptyset \mid \emptyset \mid \{(\texttt{main}, s, [], [])\}\rangle$ corresponds to a machine expression s and therefore the measure can also be applied to machine expressions.

$$spmin_M(S) = \min\{sps_M(Red) \mid Red \text{ is a terminating evaluation sequence starting with } S\}$$

Sequences of #upd(.) on the stack, e.g. #upd(x) : #upd(y) : #upd(z) : $\mathcal{S}$, lead to a sequence of copyings. This can be seen as a `letrec`-environement, for the example we then have `letrec` $x = y, y = z, z =$ `True`, that leads to `letrec` $x = y, y =$ `True`, $z =$ `True` and finally to `letrec` $x =$ `True`, $y =$ `True`, $z =$ `True`. The three #upd(.)-markers on the stack have the same effect. This is a contrast to LRPgc and M1 where an additional rule was needed to prevent such cases.

Thus the adapted abstract machine is defined as follows:

Definition 6.18 (Abstract Machine CIOM1$_{sp,int}$)
The abstract machine CIOM1$_{sp,int}$ is the abstract machine CIOM1, but additionally the transition rule (GC) has to be applied whenever possible.

Thus we now can show that the abstract machine CIOM1$_{sp,int}$ can be used to measure the space consumption of CHF*-programs:

Theorem 6.6 (Space Equivalence between CHF* and CIOM1$_{sp,int}$)
If a CHF*-process P is translated to its corresponding simplified process P_m, then *spmin*(P) = *spmin*$_M(P_m)$ holds, if CIOM1$_{sp,int}$ is used for the evaluation of P_m.

Proof
The translation σ (see Definition 2.29), that transforms P to P_m, is the reverse of the transformation (ucp) on the expression layer. (ucp) allows a step-by-step translation from CHF*-expressions to simplified expressions and does not affect the space consumption. On the monadic layer, the MVars are also only allowed to have variables as content and therefore σ creates a binding, referenced by the MVar, that contains the real content. This is the same approach as used for the expression layer and also does not affect the space consumption.

The garbage collection (GC) of the machine CIOM1$_{sp,int}$ is applied as often as possible. For CHF* the garbage collection (gc) is never applied during the standard reduction itself, but it is applied before a space-value is taken into account by *sps*. Also the measure *sps*$_M$ only takes garbage-free states into account. Hence in summary the space measurement for CHF* and CIOM1$_{sp,int}$ is the same w.r.t. garbage collection.

For an easier analysis of differences in evaluation and therefore space consumption, we put the abstract machine states into relation with the corresponding CHF*-processes:

CHF*-bindings are represented as usual heap-bindings in $\mathcal{H}$, MVars are aggregated in $\mathcal{M}$ and threads are in $\mathcal{T}$. Hence the structure of CHF*-processes differs from CIOM1$_{sp,int}$-states, but the space consumption is identical. Thus we only need to consider the differences w.r.t. evaluation.

For most of the functional and monadic transformations there is no difference. A transformation with potential of a difference in space consumption is (Update): For a variable-to-variable-chain, CHF* copies a value to each intermediate position of the chain and this is the same effect caused by sequences of #upd(.) on the stack. Moreover also (seq) and (case) require the needed value or constructor application to be copied to the argument of the corresponding `seq`- or `case`-expression, the same effect is caused by an (Update) on the CIOM1$_{sp,int}$.

The CHF*-rules (cpcxa) and (cpcxb) also cause no difference in space consumption: (cpcxa) is used to allow sharing of constructor applications. This is already achieved by σ and obviously does not affect the space consumption. (cpcxb) also occurs on the CIOM1$_{sp,int}$, since for simplified expressions all constructor applications only have variables as arguments and therefore in this situation a (Lookup) followed by an immediate (Update) is triggered and leads to the same space consumption.

In summary this shows the claim.

6.3.2 Abstract Machine CIOM1$_{sp,par}$

The abstract machine CIOM1$_{sp,int}$ as defined in Definition 6.18 can be used for space analyses using a sequential standard reduction. We adapt the CIOM1$_{sp,int}$, in order that parallel reduction steps are possible and also all calculation paths are considered that affect the space measurement. The result is the abstract machine CIOM1$_{sp,par}$.

For each transition step, the CIOM1$_{sp,int}$ uses a scheduler to choose a single thread that proceeds, for example a round-robin-approach (e.g. see (ADAD18)) can be used where the time-slices are random numbers to obtain fairness. The CIOM1$_{sp,par}$ in contrast should be able to proceed in many processes in parallel in a single transition step. Also we want to find the space-minimum, hence in case of nondeterminism all evaluation paths need to be considered.

To implement this parallelism, we use another perspective: A thread that is able to proceed can be *delayed* by the scheduler, but at least one thread must be able to proceed and not delayed by the scheduler, otherwise the current CIOM1$_{sp,par}$-state would not change in the transition-step. The motivation to use delays instead of a set of threads that are forced to proceed in parallel, is the fact, that a CIOM1$_{sp,par}$-state where all threads proceed in parallel has the best runtime properties, especially if we take the complexity of the overall problem into account. The abstract machine can be both used for runtime and space analyses, however it is often crucial to delay some thread – even if it might be better w.r.t. runtime not to delay a thread – to gain a better space consumption.

The abstract machine CIOM1$_{sp,par}$ is an adaption of CIOM1$_{sp,int}$. Since an evaluation path with minimal space consumption is calculated, we need to keep all possibilities w.r.t. nondeterminism in mind and also the parallelism using delays requires this. Hence the states of the CIOM1$_{sp,par}$ are slightly modified compared to CIOM1$_{sp,int}$, where the possible paths are modeled by a list of CIOM1$_{sp,int}$-states. *checksum*(S) calculates a checksum for state S, to allow the removing of equal states later.

Definition 6.19 (CIOM1$_{sp,par}$ State)
We interpret our model as tree. A state of CIOM1$_{sp,par}$ (node of the tree) is the pair $([(S, sps_C, cs, d)], l)$, where S is a CIOM1$_{sp,int}$-state, sps_C the maximal `cmsize`-value of garbage free states on the path from the root to S, *cs* is a checksum to identify S without considering sps_C, d the depth in the tree and l is either undefined (denoted as –) or a tuple of a leaf with minimal sps_C-value, checksum and depth.

The definition of initial states is straightforward:

Definition 6.20 (CIOM1$_{sp,par}$ Initial State)
Given a machine expression s, the corresponding CIOM1$_{sp,par}$ initial state is:

$$([(S, \texttt{size}(s), \mathit{checksum}(S), 0)], -) \quad \text{with } S = \langle \varnothing \mid \varnothing \mid \{\ (\texttt{main}, s, [], [])\ \} \rangle$$

The delays are defined for CIOM1$_{sp,int}$-states as follows. Since CIOM1$_{sp,par}$-states are a list of such states, the definition also holds for CIOM1$_{sp,par}$-states.

Definition 6.21 (Delay for Threads)
Given a CIOM1$_{sp,int}$-state $S = \langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T} \rangle$. Let $\mathcal{T}_P$ be a subset of $\mathcal{T}$ containing all threads that are able to proceed.
A *delay* is a mapping from all thread-names of $\mathcal{T}$ to Booleans, where a thread is only delayed if its corresponding Boolean value is `True`. For the threads in $\mathcal{T}_P$ it is required that at least one thread is not delayed.
A delay is written as $(t_1 \mapsto b_1, \ldots, t_n \mapsto b_n)$ where t_i is the name of the thread and b_i the corresponding Boolean value.
A *delay-function* takes a CIOM1$_{sp,int}$-state and calculates a set of delays. Note that this set is not required to contain all delays.

We give an example for delays:

Example 6.3 (Delays)
Consider the following CIOM1$_{sp,int}$-state:

$$\langle \mathcal{H} \mid \mathcal{M}, \{z\,\mathbf{m}\,\texttt{Nil}\} \mid \{(\texttt{main}, \texttt{takeMVar}\ z, [], []), (\texttt{t1}, \texttt{False}, [\texttt{\#upd}(x)], []), \\ (\texttt{t2}, \texttt{False}, [\texttt{\#upd}(y)], [])\}\rangle$$

All threads are able to proceed and there is also no interference between the threads. Thus we have the following delays:

$$\begin{array}{l}
(\texttt{main} \mapsto \texttt{False},\ \texttt{t1} \mapsto \texttt{False},\ \texttt{t2} \mapsto \texttt{True}\ \ )\\
(\texttt{main} \mapsto \texttt{False},\ \texttt{t1} \mapsto \texttt{True}\ \ ,\ \texttt{t2} \mapsto \texttt{False})\\
(\texttt{main} \mapsto \texttt{False},\ \texttt{t1} \mapsto \texttt{True}\ \ ,\ \texttt{t2} \mapsto \texttt{True}\ \ )\\
(\texttt{main} \mapsto \texttt{True}\ \ ,\ \texttt{t1} \mapsto \texttt{False},\ \texttt{t2} \mapsto \texttt{False})\\
(\texttt{main} \mapsto \texttt{True}\ \ ,\ \texttt{t1} \mapsto \texttt{False},\ \texttt{t2} \mapsto \texttt{True}\ \ )\\
(\texttt{main} \mapsto \texttt{True}\ \ ,\ \texttt{t1} \mapsto \texttt{True}\ \ ,\ \texttt{t2} \mapsto \texttt{False})
\end{array}$$

Note that $(\texttt{main} \mapsto \texttt{True},\ \texttt{t1} \mapsto \texttt{True},\ \texttt{t2} \mapsto \texttt{True})$ is not a valid delay.
Thus we calculate all Boolean combinations of all threads that can proceed except of the case where all threads would be delayed.

We often need the following terms for threads.

Definition 6.22 (Conflicting, Independent and Paused Threads)
In the following needed terms w.r.t. threads are defined:

1. We say that the threads $t_1, \ldots, t_n$ are *in conflict* or *conflicting threads* if they all can proceed and want to perform the same operation on an MVar, i.e. all threads want to perform (TakeMVar) or (PutMVar) on the same MVar and the operation is possible.
2. The threads $t_1, \ldots, t_n$ are *independent* if none of them is in conflict with any of the other threads.
3. If a thread *pauses*, then it remains completely unchanged.

We give an algorithm that calculates a list of successor states of a CIOM1$_{sp,int}$-state.

Intuitively each thread that can perform (unIO) or (fork) is allowed to proceed. Then delays are used to determine which threads proceed. All threads that are independent from each other proceed if the current delay permits this, for conflicting threads additional states need to be generated to handle all combinations.

Algorithm 6.4 (SuccState Successor States of CIOM1$_{sp,int}$-State)
Let $S = \langle\mathcal{H} \mid \mathcal{M} \mid \mathcal{T}\rangle$ be a CIOM1$_{sp,int}$-state and we assume that also a delay-function is given. Calculate a list containing triples $(S_i, sps_{C,i}, cs_i)$ for each successor state S_i with the corresponding sps_C-value and checksum as follows:

1. Let $\mathcal{T}_{uf}$ be a subset of $\mathcal{T}$ containing all threads, where either (unIO) or (fork) is applicable (compare Definition 2.44). Let $\mathcal{T}'_{uf}$ be the threads after all of the threads of $\mathcal{T}_{uf}$ performed its (unIO)- or (fork)-operation.
2. Let $\mathcal{T}_w$ be a subset of $\mathcal{T}$ containing all threads that have to wait (i.e. threads that want to perform (PutMVar) on a non-empty MVar or (TakeMVar) on an empty MVar) and let $S_p := \langle\mathcal{H} \mid \mathcal{M} \mid \mathcal{T}_p\rangle$ where $\mathcal{T}_p = \mathcal{T}\backslash(\mathcal{T}_{uf} \cup \mathcal{T}_w)$.
3. Use the delay-function to calculate all delays for S_p. For each delay construct corresponding successor states: All threads in S_p cannot perform (unIO) or (fork), since these threads were filtered above.
   (I) For all threads that are independent from each other apply (IOM1) simultaneously, yielding the state $S'_p = \langle\mathcal{H}' \mid \mathcal{M}' \mid \mathcal{T}'\rangle$. Since there are no conflicts, $\mathcal{H}'$ and $\mathcal{M}'$ can be calculated straightforwardly.
   (II) Group the conflicting threads of S'_p by the MVar-names, they want to access: $G = \{\ldots\ (t_{i,1}, \ldots, t_{i,n_i})\ \ldots\}$ where $t_{i,j}$ is the j-th thread of group i.
   Since all threads of the same group want to access the same MVar, only a single thread of a group is able to proceed simultaneously. Go trough each of these groups $(t_{i,1}, \ldots, t_{i,n_i})$ one after each other: For each combination, where only one of the threads of the current group proceeds, apply the approach of (I) yielding a corresponding state only containing the threads of the corresponding group. Also remember the changes to heap and MVars for each state.
   Use the previously calculated states together with the changes to heap and MVars to calculate all combinations of the states between the groups (i.e. a generalized Cartesian product). Since for each state the changes to heap and MVars were memorized, the heaps and MVars can be constructed by applying the changes to heap and MVars to the ones from S'_p.
4. For each state $\langle\mathcal{H}'' \mid \mathcal{M}'' \mid \mathcal{T}''\rangle$ calculated in the last step add $\mathcal{T}'_{uf}$ and $\mathcal{T}_w$ yielding $\langle\mathcal{H}'' \mid \mathcal{M}'' \mid \mathcal{T}'' \cup \mathcal{T}'_{uf} \cup \mathcal{T}_w\rangle$.
5. Gather all states from the last step in form of a tuples together with the sps_C-values and checksums into a list.

Lemma 6.6 (Correctness of SuccState)
Algorithm 6.4 calculates the successor states of a CIOM1$_{sp,int}$-state using a delay-function.

Proof
Threads performing (unIO) or (fork) do not affect heap or MVars, hence they can be handled separated from the other threads.

For every delay that is calculated by the delay-function the independent threads can directly proceed. For conflicting threads a grouping by the MVar, the threads want to access, allows to calculate all needed combinations: Since in each group every thread tries to access the same MVar, only one per group can proceed. Thus we need to calculate all combinations per group, where only one thread proceeds and then the groups are combined by merging each state of a group with each of the other groups. Since the changes of the heap and MVars are independent between the groups, the changes of MVars and heaps can be merged without conflicts.

However it is crucial to perform the operations for heap and MVars correctly. This can be done using a recording-approach: For each thread record the changes on heap and MVars and in the end apply all of the changes one after each other. This leads to correct heap and MVars.

We give an example for the computation of successor states:

Example 6.4 (Computation using SuccState)
We focus on the CIOM1$_{sp,int}$-states. Consider the following state:

$$\begin{aligned}\langle \mathcal{H}, x = \texttt{True}, y = \texttt{False}, k = \texttt{Nil} \mid \mathcal{M}, \{i\,\mathbf{m}\,-, j\,\mathbf{m}\,-\} \mid \{&(\texttt{main}, i, [], [\#\text{take}]),\\ &(\texttt{t1}, i, [], [\#\text{put}(x)]),\\ &(\texttt{t2}, i, [], [\#\text{put}(y)]),\\ &(\texttt{t3}, i, [], [\#\text{take}]),\\ &(\texttt{t4}, \texttt{future}\ k, [], []),\\ &(\texttt{t5}, j, [], [\#\text{put}(x)]),\\ &(\texttt{t6}, j, [], [\#\text{put}(y)]) \quad \}\rangle\end{aligned}$$

We apply Algorithm 6.4. First of all we have $\mathcal{T}_{uf} = \{(\texttt{t4}, \texttt{future}\ k, [], [])\}$ and a single application of (fork) yields $\mathcal{T}'_{uf} = \{(\texttt{t4}, \texttt{return t7}, [], []), (\texttt{t7}, k, [], [])\}$.
Then we have $\mathcal{T}_w = \{(\texttt{main}, z, [], [\#\text{take}]), (\texttt{t3}, i, [], [\#\text{take}])\}$, since i is currently empty. Removing all threads of $\mathcal{T}_{uf}$ and $\mathcal{T}_w$ leads to the following state S_p:

$$\begin{aligned}&\langle \mathcal{H}, x = \texttt{True}, y = \texttt{False}, k = \texttt{Nil} \mid \mathcal{M}, \{i\,\mathbf{m}\,-, j\,\mathbf{m}\,-\} \mid\\ &\quad \{(\texttt{t1}, i, [], [\#\text{put}(x)]), (\texttt{t2}, i, [], [\#\text{put}(y)]),\\ &\quad\ (\texttt{t5}, j, [], [\#\text{put}(x)]), (\texttt{t6}, j, [], [\#\text{put}(y)]) \quad \}\rangle\end{aligned}$$

We assume that the delay-function delays none of the threads, i.e. the delay-function only yields a single combination.
For the threads that need to be considered now, everyone is in conflict: There are two groups of threads in conflict for the same MVar, i.e. $G = \{(\texttt{t1}, \texttt{t2}), (\texttt{t5}, \texttt{t6})\}$. For the group $(\texttt{t1}, \texttt{t2})$ we have the combination where t1 proceeds and t2 pauses and vice versa, hence the following threads with the memory what has to be done w.r.t. the MVars later:

$$(\texttt{t1}, \texttt{return}\ (), [], []), (\texttt{t2}, i, [], [\#\text{put}(y)])$$
where x needs to be written to MVar i later

$$(\texttt{t1}, i, [], [\#\text{put}(x)]), (\texttt{t2}, \texttt{return}\ (), [], [])$$
where y needs to be written to MVar i later

For the group $(\mathtt{t5}, \mathtt{t6})$ we have the combination where `t5` proceeds and `t6` pauses and vice versa, hence the following threads with the memory what has to be done w.r.t. the MVars later:

$$\begin{array}{l}(\mathtt{t5}, \mathtt{return}\ (), [], []), (\mathtt{t6}, j, [], [\#\mathrm{put}(y)]) \\ \qquad\qquad \text{where } x \text{ needs to be written to MVar } j \text{ later} \\ (\mathtt{t5}, j, [], [\#\mathrm{put}(x)]), (\mathtt{t6}, \mathtt{return}\ (), [], []) \\ \qquad\qquad \text{where } y \text{ needs to be written to MVar } j \text{ later}\end{array}$$

The combinations per group and merge of the groups are as follows:

$$\begin{array}{l}(\mathtt{t1}, \mathtt{return}\ (), [], []), (\mathtt{t2}, i, [], [\#\mathrm{put}(y)]), \\ (\mathtt{t5}, \mathtt{return}\ (), [], []), (\mathtt{t6}, j, [], [\#\mathrm{put}(y)]) \\ \qquad\qquad \text{where } x \text{ needs to be written to MVars } i \text{ and } j \text{ later} \\ (\mathtt{t1}, \mathtt{return}\ (), [], []), (\mathtt{t2}, i, [], [\#\mathrm{put}(y)]), \\ (\mathtt{t5}, j, [], [\#\mathrm{put}(x)]), (\mathtt{t6}, \mathtt{return}\ (), [], []) \\ \qquad\qquad \text{where } x \text{ needs to be written to MVar } i \text{ and } y \text{ to MVar } j \text{ later} \\ (\mathtt{t1}, i, [], [\#\mathrm{put}(x)]), (\mathtt{t2}, \mathtt{return}\ (), [], []), \\ (\mathtt{t5}, \mathtt{return}\ (), [], []), (\mathtt{t6}, j, [], [\#\mathrm{put}(y)]) \\ \qquad\qquad \text{where } y \text{ needs to be written to MVar } i \text{ and } x \text{ to MVar } j \text{ later} \\ (\mathtt{t1}, i, [], [\#\mathrm{put}(x)]), (\mathtt{t2}, \mathtt{return}\ (), [], []), \\ (\mathtt{t5}, j, [], [\#\mathrm{put}(x)]), (\mathtt{t6}, \mathtt{return}\ (), [], []) \\ \qquad\qquad \text{where } y \text{ needs to be written to MVars } i \text{ and } j \text{ later}\end{array}$$

Heap and MVars now can be calculated yielding states and then finally $\mathcal{T}_{uf}$ and $\mathcal{T}_w$ need to be added to the four states:

$$\begin{array}{l}[\langle \mathcal{H}, x = \mathtt{True}, y = \mathtt{False}, k = \mathtt{Nil} \mid \mathcal{M}, \{i\,\mathbf{m}\,x, j\,\mathbf{m}\,x\} \mid \\ \quad \{(\mathtt{main}, i, [], [\#\mathrm{take}]), (\mathtt{t1}, \mathtt{return}\ (), [], []), (\mathtt{t2}, i, [], [\#\mathrm{put}(y)]), \\ \quad\ (\mathtt{t3}, i, [], [\#\mathrm{take}]), (\mathtt{t4}, \mathtt{return}\ \mathtt{t7}, [], []), (\mathtt{t5}, \mathtt{return}\ (), [], []), \\ \quad\ (\mathtt{t6}, j, [], [\#\mathrm{put}(y)]), (\mathtt{t7}, k, [], []) \qquad\qquad\qquad\qquad \}\rangle, \\ \langle \mathcal{H}, x = \mathtt{True}, y = \mathtt{False}, k = \mathtt{Nil} \mid \mathcal{M}, \{i\,\mathbf{m}\,x, j\,\mathbf{m}\,y\} \mid \\ \quad \{(\mathtt{main}, i, [], [\#\mathrm{take}]), (\mathtt{t1}, \mathtt{return}\ (), [], []), (\mathtt{t2}, i, [], [\#\mathrm{put}(y)]), \\ \quad\ (\mathtt{t3}, i, [], [\#\mathrm{take}]), (\mathtt{t4}, \mathtt{return}\ \mathtt{t7}, [], []), (\mathtt{t5}, j, [], [\#\mathrm{put}(x)]), \\ \quad\ (\mathtt{t6}, \mathtt{return}\ (), [], []), (\mathtt{t7}, k, [], []) \qquad\qquad\qquad\qquad \}\rangle, \\ \langle \mathcal{H}, x = \mathtt{True}, y = \mathtt{False}, k = \mathtt{Nil} \mid \mathcal{M}, \{i\,\mathbf{m}\,y, j\,\mathbf{m}\,x\} \mid \\ \quad \{(\mathtt{main}, i, [], [\#\mathrm{take}]), (\mathtt{t1}, i, [], [\#\mathrm{put}(x)]), (\mathtt{t2}, \mathtt{return}\ (), [], []), \\ \quad\ (\mathtt{t3}, i, [], [\#\mathrm{take}]), (\mathtt{t4}, \mathtt{return}\ \mathtt{t7}, [], []), (\mathtt{t5}, \mathtt{return}\ (), [], []), \\ \quad\ (\mathtt{t6}, j, [], [\#\mathrm{put}(y)]), (\mathtt{t7}, k, [], []) \qquad\qquad\qquad\qquad \}\rangle, \\ \langle \mathcal{H}, x = \mathtt{True}, y = \mathtt{False}, k = \mathtt{Nil} \mid \mathcal{M}, \{i\,\mathbf{m}\,y, j\,\mathbf{m}\,y\} \mid \\ \quad \{(\mathtt{main}, i, [], [\#\mathrm{take}]), (\mathtt{t1}, i, [], [\#\mathrm{put}(x)]), (\mathtt{t2}, \mathtt{return}\ (), [], []), \\ \quad\ (\mathtt{t3}, i, [], [\#\mathrm{take}]), (\mathtt{t4}, \mathtt{return}\ \mathtt{t7}, [], []), (\mathtt{t5}, j, [], [\#\mathrm{put}(x)]), \\ \quad\ (\mathtt{t6}, \mathtt{return}\ (), [], []), (\mathtt{t7}, k, [], [])\} \qquad\qquad\qquad\qquad \}\rangle]\end{array}$$

It is easy to see that the checksums and sps_C-values can be calculated straightforwardly.

The overall calculation is an iterative depth-first-search and enables to cut parts of the tree. Especially if the tree has a greater depth on the right-side, a breadth-first search would have more possibilities of cutting parts of a tree very fast. However the space consumption is very high and a practical limitation. Moreover it helps to find a leaf as early as possible, since leafs are important for tree cuts and in many cases depth-first-search finds leafs earlier with less space consumption than breadth-first search – then this helps to cut parts of the tree in the ongoing calculation.

A final state is a state that definitely provides the target sps_C-value and directly models the idea that a tree does not need to be calculated entirely:

Definition 6.23 (CIOM1$_{sp,par}$ Final State)
A CIOM1$_{sp,par}$-state S is a *final state*, if S is of one of the following forms:

1. $([], (S_{leaf}, sps_{C,leaf}, cs_{leaf}, d_{leaf}))$
2. $([(S_1, sps_{C,1}, cs_1, d_1), \dots, (S_n, sps_{C,n}, cs_n, d_n)],$
   $(S_{leaf}, sps_{C,leaf}, cs_{leaf}, d_{leaf}))$
   where $sps_{C,leaf} \leq sps_{C,i}$ holds for $i \in \{1, \dots, n\}$.

Since a maximal depth is now used for the depth-first-search we also define failed states, where the maximal depth terminated the calculation, but the state does not provide a result. If a failed state is reached, then the maximal depth is increased and the whole calculation starts again.

Definition 6.24 (CIOM1$_{sp,par}$ Failed State)
A CIOM1$_{sp,par}$-state S is a *failed state*, if S is of the following form, where a $sps_{C,i}$ exists with $sps_{C,i} < sps_{C,leaf}$ and $d_j \geq d_{max}$ holds for all $j \in \{1, \dots, n\}$ and a given maximal depth d_{max}:

$$([(S_1, sps_{C,1}, cs_1, d_1), \dots, (S_n, sps_{C,n}, cs_n, d_n)], (S_{leaf}, sps_{C,leaf}, cs_{leaf}, d_{leaf}))$$

Checksums are used to eliminate CIOM1$_{sp,int}$-states that are included in CIOM1$_{sp,par}$-states. Also alpha equivalence of CIOM1$_{sp,int}$-states can be used to reduce the size of the tree.

Definition 6.25 (α-Equivalence of CIOM1$_{sp,int}$-States)
Two CIOM1$_{sp,int}$-states S_1 and S_2 are *α-equivalent*, if there exists a renaming σ of bound variables so that $S_1 = \sigma(S_2)$ holds.
Note that also names of threads are allowed to be renamed by σ, an exception is the main-thread.

Often some delays only lead to minor differences, that cause states where some threads are exchanged but semantically lead to the same result. This might lead to α-equivalent states if the different combinations of the threads between the compared states are tested. As excepted this increases the runtime.

If not only the top-`letrec`, but also the `letrec`-subexpressions are free of garbage bindings, then the order of the heap-bindings can be calculated, otherwise also all combinations need to be checked. Since garbage collection has a strong impact on sps_C it is not correct to apply (GC) to all `letrec`-expressions of a state before testing the α-equivalence. We give a definition for this strict kind of garbage-freeness.

Definition 6.26 (GC-Normalform of CIOM1$_{sp,int}$-States)
A CIOM1$_{sp,int}$-state S is in *GC-normalform*, if the complete state is free of garbage, i.e. (GC) does not apply and also there are no `letrec`-expressions as subexpressions that contain garbage.

Since we are only interested in unneeded `letrec`-bindings and MVars, it can be tested efficiently if a CIOM1$_{sp,int}$-state is in GC-normalform.

Proposition 6.5 (Complexity of GC-Normalform-Testing of CIOM1$_{sp,int}$-State)
For a CIOM1$_{sp,int}$-state it can be tested in time $O(n \log n)$ whether the state is in GC-normalform, where n is the syntactical size of the state.

Proof
The usual garbage collector can be extended straightforwardly without increasing the asymptotic runtime.

The complexity to test the α-equivalence of states is as follows, where the complexity given in Proposition 6.5 helps to automatically choose the most efficient strategy:

Proposition 6.6 (Complexity of α-Equivalence-Testing of CIOM1$_{sp,int}$-States)
Let S_1 and S_2 be CIOM1$_{sp,int}$-states.

1. If S_1 and S_2 have a different structure, then the α-equivalence-property can be falsified in $O(n)$, where n is the syntactical size of the smaller state.
2. If S_1 and S_2 have not a different structure, then the syntactical size and number of threads is identical and there are two cases, where in both cases N is the number of threads and n the syntactical size of the states:
   (I) If S_1 and S_2 are in GC-normalform, then the α-equivalence-test of the two states can be performed in time $O(N! \cdot (n \log n))$.
   (II) If not both S_1 and S_2 is in GC-normalform, then the α-equivalence-test of the two states can be performed in time $O(N! \cdot n!)$.

Proof
The structural test on syntactical level is straightforward and the complexity is caused by the fact, that the whole comparison between the two states can be cancelled if a single difference is found.

The stacks and IO-stacks can be reverted to the expressions. Note that there cannot be an overlap between two threads and the same heap variable or MVar, since then one of those threads would have to wait. Then a part of the whole α-equivalence-test can be performed on expression-level. However all orderings of threads need to be considered.

If the `letrec`-expressions are all free of garbage, then the heap-bindings can be implicitly ordered by a numbering-approach. Maps can be used to assign the numbers to variables. All permutations need to be considered for the threads. Hence for the case of garbage-free `letrec`-expressions, the time complexity is $O(N! \cdot (n \log n))$.

If the `letrec`-expressions are not free of garbage, then all permutations also for `letrec`-bindings and heaps need to be considered, hence the time is in $O(N! \cdot n!)$.

The best runtime is achieved by first running the structural test, then check if both states are in GC-normalform and depending on the result applying the most efficient strategy.

A transition step of the machine CIOM1$_{sp,par}$ uses Algorithm 6.4 and implements a variant of an iterative depth-first-search:

Definition 6.27 (CIOM1$_{sp,par}$ Transition Step)
A transition step from one CIOM1$_{sp,par}$-state to the next one is defined by the following algorithm:
Given a CIOM1$_{sp,par}$-state

$$S = ([(S_1, sps_{C,1}, cs_1, d_1), \dots, (S_n, sps_{C,n}, cs_n, d_n)], T)$$
$$\text{where } T = - \text{ or } T = (S_{leaf}, sps_{C,leaf}, cs_{leaf}, d_{leaf})$$

that is neither a final state nor a failed state, hence the list is not empty. Let d_{max} be the maximal depth and we write L for the first component of S.

1. If $d_1 > d_{max}$, then remove $(S_1, sps_{C,1}, cs_1, d_1)$ from S, yielding S_d and return S_d as overall result.
2. Apply Algorithm 6.4 to S_1 and extend all tuples of the result by a fourth component containing $d_1 + 1$. This step yields the list L_1.
3. Apply (GC) to every state in L_1 yielding $L_{1,gc}$.
4. For every state S_j in $L_{1,gc}$, if S_j is a CIOM1$_{sp,int}$-final-state:
   If $T = -$ or S_j has a lower sps_C-value than $sps_{C,leaf}$, then set $S_{leaf} = S_j$ and remove every tuple from L that has a higher or equal sps_C-value than $sps_{C,leaf}$, yielding L'. Also update $sps_{C,leaf}$, cs_{leaf} and d_{leaf} accordingly.
5. If $T \neq -$: Remove every tuple from $L_{1,gc}$ that has a higher or equal sps_C-value than $sps_{C,leaf}$, yielding $L'_{1,gc}$.
6. Remove every tuple from $L'_{1,gc}$, where a state with the same checksum or being α-equivalent in L' can be found with lower or equal sps_C value, yielding $L''_{1,gc}$.
7. Add $L''_{1,gc}$ in front of L' yielding the result list L_{res}. Return (L_{res}, S_{leaf}).

The removing of states depending on a so far found leaf is correct:

Lemma 6.7 (Correctness of Tree Cuts)
Let $L = [(S_1, sps_{C,1}), \dots, (S_n, sps_{C,n})]$ where S_i are CIOM1$_{sp,int}$-states and $sps_{C,i}$ the corresponding sps_C-values. Also we assume that a CIOM1$_{sp,int}$-final-state S_{leaf} is known, where $sps_{C,leaf}$ is the corresponding sps_C-value.

Removing all S_i with $sps_{C,i} \geq sps_{C,leaf}$ does not affect the overall result.

Proof
Let w.l.o.g. $P_1 = [S, S_{1\text{-}1}, S_{1\text{-}2} \dots, S_{1\text{-}n}]$ be the path in the tree from the root S to the leaf $S_{1\text{-}n} = S_{leaf}$. Also let $P_k = [S, S_{k\text{-}1}, S_{k\text{-}2}]$ be the states on the path in the tree from the root to the state $S_{k\text{-}2}$.

$sps_{C,i-j}$ denotes the sps_C-value of state $S_{i\text{-}j}$. Let $sps_{C,k\text{-}2} \geq sps_{C,1\text{-}n}$. The maximal space value of P_1 is equal to $sps_{C,1\text{-}n}$ and the maximal space value of P_2 is at least

$sps_{C,k\text{-}2}$. The overall required space over all paths of the complete tree is at most $sps_{C,1\text{-}n}$.

Hence even if all states following $S_{k\text{-}2}$ have smaller space values, the overall result is not affected by P_k, therefore the states following $S_{k\text{-}2}$ do not need to be considered anymore.

$$\begin{tikzcd}
 & & & & & & S \arrow[dll] \arrow[drrrr] & & & & & \\
 & & & & S_{1\text{-}1} \arrow[dll] \arrow[dr] & & \cdots & & & & S_{k\text{-}1} \arrow[dll] \arrow[dr] & \\
 & & S_{1\text{-}2} \arrow[dl] \arrow[dr] & & & \vdots & & & S_{k\text{-}2} \arrow[dl] \arrow[dr] & & & \vdots \\
 & \vdots \arrow[dl] \arrow[dr] & & \vdots & & & & \vdots & & \vdots & & \\
 S_{1\text{-}n} & & \vdots & & & & & & & & &
\end{tikzcd}$$

The optimization using checksums and α-equivalence is also correct:

Lemma 6.8 (Correctness of State-Eliminations Using Checksums and α-Equivalence)
Let S_i be a CIOM1$_{sp,int}$-state with checksum cs_i and sps_C-value $sps_{C,i}$. Given a CIOM1$_{sp,par}$-state containing a CIOM1$_{sp,int}$-state S_p with checksum cs_p and sps_C-value $sps_{C,p}$. Also $sps_{C,i} \geq sps_{C,p}$ holds.
If the checksums are equal (i.e. $cs_i = cs_p$ holds) or S_i and S_p are α-equivalent, then the overall result is the same, both if S_i is added to the stack or not.

Proof
If two CIOM1$_{sp,int}$ states have the same checksum or are α-equivalent, then they at most differ w.r.t. names of bound variables or threads, but are semantic equivalent. Hence both for S_i and S_p the further calculation steps have exactly the same space consumption.

Since $sps_{C,i} \geq sps_{C,p}$, the so far overall space consumption using S_i instead of S_p is higher or equal and therefore the removing of S_i does not affect the overall optimal solution.

The returned space-optimal leaf is not guaranteed to be the runtime-optimal under the space-optimal leafs.

Proposition 6.7 (Returned Leaf using Tree Cuts not Necessarily Runtime-Optimal)
If the optimizations w.r.t. tree cuts are not performed, then a set of leafs $\{L_1, \ldots, L_n)$ can be calculated, that contains all leafs of the state tree.
In contrast to this set, let L_{tc} be the leaf that is calculated using tree cuts.
Then there might be a leaf L_i that has the same sps_C-value but a better runtime than L_{tc}, that cannot be found using tree cuts.

Proof
Consider the following tree:

$$\begin{tikzcd}
 & & & & & S \arrow[dll] \arrow[drr] & & & & \\
 & & & \vdots \arrow[dll] \arrow[dr] & & \cdots & & \vdots \arrow[dll] \arrow[d] \arrow[drr] & & \\
 & \vdots \arrow[dl] \arrow[dr] & & & \vdots & \vdots & & S_{2\text{-}(n-2)} \arrow[dl] \arrow[dr] & & \vdots \\
S_{1\text{-}(n-1)} \arrow[d] \arrow[dr] & & \vdots & & & & \vdots & & S_{2\text{-}(n-1)} & \\
S_{1\text{-}n} & \vdots & & & & & & & &
\end{tikzcd}$$

In this tree $S_{1\text{-}n}$ and $S_{2\text{-}(n-1)}$ are leafs, where n denotes the depth. We assume that the sps_C-values of $S_{1\text{-}n}$, $S_{2\text{-}(n-1)}$ and $S_{2\text{-}(n-2)}$ are equal.

Because of the depth-first-search, $S_{1\text{-}n}$ is found before the state $S_{2\text{-}(n-2)}$ is processed. Hence the tree cut mechanism (see Definition 6.27) ignores the leaf $S_{2\text{-}(n-1)}$, since the state $S_{2\text{-}(n-2)}$ is not added to the stack. However the leaf $S_{2\text{-}(n-1)}$ has the same space consumption and is one level higher than $S_{1\text{-}n}$. This general approach can be applied for different measurements of runtime.

A further idea to reduce the overall runtime is the extension of the optimization using checksums and α-equivalence:

Before adding a state to the stack, it already is checked whether there exists a state with the same checksum or an α-equivalent state. Also it is possible to remove elements from the stack, i.e. applying this approach in the other direction.

Assume that a CIOM1$_{sp,int}$-state S_j is intended to be added to the stack (compare Definition 6.27) and there is a CIOM1$_{sp,int}$-state S_k on the stack that has the same checksum or is α-equivalent. It is correct to remove S_k using the same arguments as in the proof of Lemma 6.7, but since currently S_j is currently processed and not S_k or one of the following states of S_k, S_k must be on a path of the tree that is further to the right than S_j.

Hence if the there is a leaf in the sub-tree of S_j, then the depth-first-search finds it and then – depending on the space-consumption – eliminates S_k. Thus the approach to eliminate states on the stack before adding states, that are α-equivalent or have the same checksum, decreases the space consumption of a practical implementation of the abstract machine by a low amount but heavily increases the runtime, since the stack might contain many states for an exponential tree and this leads to additional iterations with a high frequency.

In the following we analyze the impact of delays and give a useful set of delays to calculate the space-optimum. Delays can increase the amount of work:

Proposition 6.8 (Delays May Duplicate Work)
A set of delays might introduce duplicated states.

Proof
Consider the following state CIOM1$_{sp,int}$-state:

$$\begin{aligned}\langle \mathcal{H}, x = \texttt{True}, y = \texttt{False} \mid \mathcal{M}, \{z\,\mathbf{m}\,-\} \mid \{&(\texttt{main}, z, [], [\#\text{take}]),\\ &(\texttt{t1}, z, [], [\#\text{put}(x)]),\\ &(\texttt{t2}, z, [], [\#\text{put}(y)])\}\rangle\end{aligned}$$

The two threads `t1` and `t2` both want to put something to the same empty MVar. If we then use the following delays

$$\{(\texttt{t1} \mapsto \texttt{False}, \texttt{t2} \mapsto \texttt{False}), (\texttt{t1} \mapsto \texttt{False}, \texttt{t2} \mapsto \texttt{True}), (\texttt{t1} \mapsto \texttt{True}, \texttt{t2} \mapsto \texttt{False})\}$$

this doubles the amount of states:

Since both threads access the same MVar one has to wait. Hence if no thread is forced to pause, i.e. $(\texttt{t1} \mapsto \texttt{False}, \texttt{t2} \mapsto \texttt{False})$, the maximal parallelization then calculates a state where `t1` proceeds while `t2` is waiting and a state where `t1` waits while `t2` proceeds – these two states are completely the same as the states calculated for $(\texttt{t1} \mapsto \texttt{False}, \texttt{t2} \mapsto \texttt{True})$ and $(\texttt{t1} \mapsto \texttt{True}, \texttt{t2} \mapsto \texttt{False})$.

Thus Proposition 6.8 requires delay-functions not to delay already conflicting threads. Moreover many transformations are not required to be delayed, because the delay of them do not affect the space consumption. This leads to the following definition of a delay-function, that can be used to perform space analyses:

Definition 6.28 (Delay-Function *DelSpaceOpt* for Space-Analyses)
Given a CIOM1$_{sp,int}$-state $S = \langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T}\rangle$. We consider the following subsets of $\mathcal{T}$:
- $\mathcal{T}_{uf}$ contains all threads that want to perform (unIO) or (fork).
- $\mathcal{T}_{w}$ contains all threads that are not able to perform a (TakeMVar)- or (PutMVar)-operation, because the corresponding MVar is empty or not filled.
- $\mathcal{T}_{c}$ contains all threads that are in conflict for MVars.
- $\mathcal{T}_{p}$ contains all other threads, i.e. $\mathcal{T} \backslash (\mathcal{T}_{uf} \cup \mathcal{T}_{w} \cup \mathcal{T}_{c})$.

The delay-function *DelSpaceOpt* follows the rules below:
1. Threads that want to perform (unIO) or (fork) are not delayed, i.e. for all $t_{uf} \in \mathcal{T}_{uf}$ we define $t_{uf} \mapsto \texttt{False}$.
2. Waiting threads are not delayed, i.e. for all $t_w \in \mathcal{T}_w$ we define $t_w \mapsto \texttt{False}$.
3. Conflicting threads are not delayed, i.e. for all $t_c \in \mathcal{T}_c$ we define $t_c \mapsto \texttt{False}$.
4. For all threads $t_p \in \mathcal{T}_p$ where the next transition rule of t_p is either an (Unwind1), (Unwind2), (Unwind3), (Unwind4), (Unwind5), (Unwind6), (Lookup), (Letrec), (Subst), (Branch), (Seq), (NewMVar), (LUnit) or (Blackhole), we define $t_p \mapsto \texttt{False}$. For (Update), (TakeMVar) and (PutMVar) delays are allowed.

We show that the delay-function *DelSpaceOpt* can be used for space analyses:

Lemma 6.9 (*DelSpaceOpt* yields Correct Space Measurement)
Using *DelSpaceOpt* as delay-function yields correct results w.r.t. space-measurement.

Proof
It is obvious that the space measurement is not affected if all delays are considered. Hence we argue that the prohibition of delays for certain transition rules does not change the overall result.

The functional transitions (Letrec), (Unwind1), (Unwind2), (Unwind3), (Unwind4), (Unwind5), (Unwind6), (Subst), (Seq), (Branch), (Lookup), (Update) cannot cause conflicts: It is no difference, which thread performs a (Lookup), since all other threads cannot continue their calculations until the value is written back to the heap using (Update) and the calculation depends on the heap-entry and not on the threads. Since all rules except of (Update) do not increase the size of the CIOM1$_{sp,int}$-state, not delaying functional transition rules cannot have an effect on space consumption for all functional transition rules except of (Update). In the case of an (Update) or (PutMVar) the size of the CIOM1$_{sp,int}$-state is increased, hence an effect on the overall result is possible by delaying. (TakeMVar) is synchronizing and thus a delay needs to be considered.

(unIO) decreases the size of the current state by 1, (fork) does not change the size. After a (fork) there is still the possibility to delay the new thread as long as needed, as if it is not created, a similar argument applies to (unIO).

Since Algorithm 6.4 already takes care of conflicting threads, i.e. for each combination the successor states are calculated, the delay of conflicting threads would only lead to duplicated threads (see Proposition 6.8). Hence it is correct not to delay conflicting threads. Also Algorithm 6.4 does not allows waiting threads to proceed, hence the abstract machine delays such threads itself and there is no difference w.r.t. the overall result caused by a delay of a waiting thread.

Now we can put the parts together and define the abstract machine:

Definition 6.29 (Abstract Machine CIOM1$_{sp,par}$)
The abstract machine CIOM1$_{sp,par}$ uses the states defined in Definition 6.19, where an initial state is defined in Definition 6.20. As delay-function *DelSpaceOpt* is used (see Definition 6.28). Until a final state (see Definition 6.23) or failed state (see Definition 6.24) is reached, Definition 6.27 is applied exhaustively using an upper bound for the depth. In case of a final state, the leaf is returned containing the required space value. In case of a failed state the upper bound for the depth is increased and the whole calculation is started again.

We now show that the abstract machine CIOM1$_{sp,par}$ can be used to measure the space consumption of CHF*-programs using parallel evaluation:

Theorem 6.7 (Space Equivalence between CHF* with Parallelization and CIOM1$_{sp,par}$)
If a CHF*-process P is translated to its corresponding simplified process P_m, then $spmin(P) = sps_C(S_{leaf})$, if the parallel evaluation is used for CHF* and S_{leaf} is the result returned by the CIOM1$_{sp,par}$-evaluation of P_m.

Proof
The only difference between CHF* in its interleaved and parallel variant, is that the parallel evaluation as defined in Definition 5.2 is used instead of the sequential evaluation. This difference between the CHF*-variants is the same difference we have between the abstract machines CIOM1$_{sp,int}$ and CIOM1$_{sp,par}$. The CIOM1$_{sp,par}$ makes the scheduling explicit and heavily uses the CIOM1$_{sp,int}$ for the real calculations w.r.t. the schedule.

For a single CIOM1$_{sp,int}$-state the algorithm SuccState calculates the following states using a delay-function. The correctness of SuccState follows from Lemma 6.6 and since the delay-function *DelSpaceOpt* is used, the correctness of space measurement follows from Lemma 6.9. Also tree cuts are performed, where Lemma 6.7 yields the correctness of space measurement and the correctness w.r.t. space measurement for the further optimizations using checksums and α-equivalence follows from Lemma 6.8. Hence the scheduling implemented by CIOM1$_{sp,par}$ yields correct space results and then for each schedule the correctness of space measurement of CIOM1$_{sp,int}$ (see Theorem 6.6) shows the claim.

The complexity of the CIOM1$_{sp,par}$ mainly depends on the algorithm SuccState, hence the number of following states in a single transition step of the CIOM1$_{sp,par}$.

Proposition 6.9 (Complexity of a Single Transition Step of CIOM1$_{sp,par}$)
Consider a single transition step of the abstract machine CIOM1$_{sp,par}$ for a state S. Let n be the syntactical size of S, c_T be the number of threads of the first state S_1 of the state-list of S, c_D be the number of threads of S_1 that might be delayed and c_S be the number of elements of the state-list of S. Let there be m groups of threads, where every thread in each group tries to perform an operation on the same MVar, then n_i for $i \in \{1, \ldots, m\}$ is the number of threads of group i.
Then the complexity of a single transition step is $O((\prod_{i=1}^{m} n_i \cdot 2^{c_D}) \cdot (c_S + c_T! \cdot n!))$ if α-equivalence is used and $O((\prod_{i=1}^{m} n_i \cdot 2^{c_D}) \cdot (n \log n + c_S))$ otherwise. In the case of $m = 0$, $\prod_{i=1}^{m} n_i$ is replaced by 1.

Proof
Given a CIOM1$_{sp,par}$-state

$$\begin{aligned} S = (& [(S_1, sps_{C,1}, cs_1, d_1), \ldots, (S_n, sps_{C,n}, cs_n, d_n)], \\ & (S_{leaf}, sps_{C,leaf}, cs_{leaf}, d_{leaf})) \end{aligned}$$

that is neither a final state nor a failed state, hence the list is not empty. Also d_1 is lower than the maximal depth.

We analyze the runtime complexity of the application of SuccState (see Algorithm 6.4) to S_1. If there are m groups of threads, where every thread in each group tries to perform an operation on the same MVar, then let $n_1, \ldots, n_m$ be the numbers of elements of each group. For each group all combinations need to be calculated where one thread proceeds while all other threads of the same group pause and then these combinations are combined with all other combinations of the other groups.

Hence for conflicting threads there are $n_1 \cdot \ldots \cdot n_m$ combinations i.e. states. Also taking the worst-case runtime of the CIOM1$_{sp,int}$-step into account, we so far have a time complexity of $O(\prod_{i=1}^m n_i \cdot \log n)$. Waiting threads and threads that intend to perform an (unIO) or (fork) do not increase the number of states. Let there be c_D threads that are able to proceed, not waiting, not in conflict with each other and all of them want to perform an (Update)-, (TakeMVar)- or (PutMVar)-operation, then the delay-function *DelSpaceOpt* yields 2^{c_D} combinations, that need to be combined with all combinations of the conflicting threads, hence the complexity increases to $O((\prod_{i=1}^m n_i \cdot 2^{c_D}) \cdot \log n)$. Then (GC) is applied to each of these states, hence we have $O((\prod_{i=1}^m n_i \cdot 2^{c_D}) \cdot (n \log n))$.

Let L be the result of SuccState. If all CIOM1$_{sp,int}$-states in L are final states and the sps_C-values are decreasing from left to right, then for each of them the whole list, i.e. first component of S, is traversed, hence the worst-case runtime for each step is $O(c_S)$. Thus this leads to $O((\prod_{i=1}^m n_i \cdot 2^{c_D}) \cdot (n \log n + c_S))$.

The further optimizations can be combined into a single traverse of L. We have the overall runtime $O((\prod_{i=1}^m n_i \cdot 2^{c_D}) \cdot (n \log n + c_S))$ if α-equivalence is not used, otherwise we have $O((\prod_{i=1}^m n_i \cdot 2^{c_D}) \cdot (c_S + c_T! \cdot n!))$ where it is assumed that the syntactical sizes of the states in L are equal to the sizes of the states of S and $n \log n$ is clearly dominated by $(c_T! \cdot n!)$ for $c_T > 0$ and therefore omitted.

The estimation is pessimistic but therefore easier to grasp, since the size n of a state is estimated roughly, because often instead of n the size of a certain CIOM1$_{sp,int}$-state is lower. The estimation helps to illustrate the complexity of the problem. However for many programs the worst-case does not occur, since *DelSpaceOpt* delays as less as possible and for a not too high amount of conflicting threads, the complexity is adequate. Also it is a good approach, to only use the testing of α-equivalence if really needed.

6.3.3 Abstract Machine CIOM1$_{t,par}$

In this section an abstract machine is defined to analyze the optimal runtime, since the runtime might be affected by a space-optimal schedule. First of all we define the runtime measure that can be used by CIOM1$_{sp,par}$.

Definition 6.30 (Reduction Length Measures mln$_P$ and mlnall$_P$)
Given a simplified process Q_m. The corresponding complete state tree T consisting of CIOM1$_{sp,int}$-states, using all possible delays and possibilities caused by conflicts on MVars, contains the leafs $L_1, \ldots, L_n$ with finite paths from root to L_i. Let $n \geq 1$.

1. For two CIOM1$_{sp,int}$-states S_j and S_{j+1} we define mln$_{P,step}(S_j)$ as 1 if at least one thread from S_j to S_{j+1} performed a (Subst)-, (Branch)- or (Seq)-step, otherwise 0. Analogous mlnall$_{P,step}(S_j)$ is defined taking all steps into account.
2. mln$_{P,C}(S_j)$ with a path $S_1, \ldots, S_j$ from root to S_j is defined as $\sum_{k=1}^{j-1}$ mln$_{P,step}(S_k)$. mlnall$_{P,C}(S_j)$ is defined analogous using mlnall$_{P,step}(S_j)$.
3. mln$_P(Q_m)$ is defined as a minimal mln$_{P,C}(L_i)$. mlnall$_P(Q_m)$ is defined as a minimal mlnall$_{P,C}(L_i)$.

If $n = 0$ then we have mln$_P(Q_m)$ = mlnall$_P(Q_m) = \infty$.

The intuition is, that a shortest path from root to leaf in a given complete state tree has the best runtime-property. However this is intuition is not correct in detail: mln_P takes each transition step of every single thread into account and therefore a low depth not guarantees that the runtime is lower than for a leaf with a higher depth. Hence we can apply the same approach as also used in $\text{CIOM1}_{sp,par}$, but the considered measure values are not the sizes, instead the so far calculated runtimes are used. Moreover there might be unneeded threads, hence it is necessary to allow delays of (Subst), (Seq) and (Branch). Also (TakeMVar) and (PutMVar) are allowed to be delayed because they synchronize. This leads to the following delay-function:

Definition 6.31 (Delay-Function *DelTimeOpt* for Runtime-Analyses)
Given a $\text{CIOM1}_{sp,int}$-state $S = \langle \mathcal{H} \mid \mathcal{M} \mid \mathcal{T} \rangle$. We consider the following subsets of $\mathcal{T}$:

- $\mathcal{T}_{uf}$ contains all threads that want to perform (unIO) or (fork).
- $\mathcal{T}_w$ contains all threads that are not able to perform a (TakeMVar)- or (PutMVar)-operation, because the corresponding MVar is empty or not filled.
- $\mathcal{T}_c$ contains all threads that are in conflict for MVars.
- $\mathcal{T}_p$ contains all other threads, i.e. $\mathcal{T} \backslash (\mathcal{T}_{uf} \cup \mathcal{T}_w \cup \mathcal{T}_c)$.

The delay-function *DelTimeOpt* follows the rules below:

1. Threads that want to perform (unIO) or (fork) are not delayed, i.e. for all $t_{uf} \in \mathcal{T}_{uf}$ we define $t_{uf} \mapsto$ `False`.
2. Waiting threads are not delayed, i.e. for all $t_w \in \mathcal{T}_w$ we define $t_w \mapsto$ `False`.
3. Conflicting threads are not delayed, i.e. for all $t_c \in \mathcal{T}_c$ we define $t_c \mapsto$ `False`.
4. For all threads $t_p \in \mathcal{T}_p$ where the next transition rule of t_p is either an (Unwind1), (Unwind2), (Unwind3), (Unwind4), (Unwind5), (Unwind6), (Lookup), (Update), (Letrec), (LUnit), (NewMVar) or (Blackhole), we define $t_p \mapsto$ `False`. For (Subst), (Branch), (Seq), (TakeMVar) and (PutMVar) delays are allowed.

We show that the delay-function *DelTimeOpt* can be used for runtime analyses:

Lemma 6.10 (*DelTimeOpt* yields Correct Time Measurement)
Using *DelTimeOpt* as delay-function yields correct results w.r.t. time-measurement.

Proof
Obviously all transformations that do not cause synchronizations or are directly counted by the measurement, cannot cause an increase of runtime and therefore it does not affect the overall runtime result not to delay these transformations. Hence we go through all transformations that might have an effect on the runtime:

It improves the runtime, if (Subst), (Branch) and (Seq) is delayed, in the case that the corresponding thread does not have an impact on the result of the main-thread (i.e. the corresponding thread can be seen as garbage). Moreover (TakeMVar) and (PutMVar) can cause synchronizations and therefore may have an impact on the overall runtime result.

Thus we can modify the abstract machine $\text{CIOM1}_{sp,par}$ to get a valid runtime measurement w.r.t. mln_P as follows.

Definition 6.32 (Abstract Machine CIOM1$_{t,par}$)
The abstract machine CIOM1$_{t,par}$ is the abstract machine CIOM1$_{sp,par}$ (see Definition 6.29) with the following modifications:

1. $\text{mln}_{P,C}$ is used instead of sps_C. In detail this applies to the states (compare Definition 6.19, Definition 6.23, Definition 6.24) and the transition step (compare Definition 6.27).
2. The delay-function *DelTimeOpt* is used instead of *DelSpaceOpt*.

We now show that the abstract machine CIOM1$_{t,par}$ can be used to calculate the minimal runtime of CHF*-programs using parallel evaluation. In the following srnrp_A is the same as srnrp_A^N (see Definition 5.3) but N is omitted since the number of processors is not limited.

Theorem 6.8 (Runtime Compatibility between CHF* with Parallelization and CIOM1$_{t,par}$)
If a CHF*-process P is translated to its corresponding simplified process P_m, then $\text{srnrp}_A(P) = \text{mln}_{P,C}(S_{leaf})$ for $A = \{(\text{lbeta}), (\text{case}), (\text{seq})\}$, if the parallel evaluation is used for CHF* and S_{leaf} is the result returned by the CIOM1$_{t,par}$-evaluation.

Proof
The delay-function *DelTimeOpt* yields correct runtime measurement, see Lemma 6.10. Moreover the abstract machine CIOM1$_{t,par}$ applies the same approach for conflicting threads and tree cuts, only using the measure $\text{mln}_{P,C}$ instead of sps_C. Tree cuts using $\text{mln}_{P,C}$ are also correct, i.e. tree cuts do not affect the overall mln_P-value. This can be shown in the same way as in the proof of Lemma 6.7. The proofs of Lemma 6.8 can be directly adapted to show that state eliminations using checksums and α-equivalence also do not affect the runtime-measurement.

Hence we now have to show that the transformation to simplified processes and also the different evaluation steps do not affect the runtime.

The translation σ (see Definition 2.29), that transforms P to P_m, is the reverse of the transformation (ucp) on the expression layer. (ucp) allows a step-by-step translation from CHF*-expressions to simplified expressions and does not affect the runtime as measured by srnrp_A and mln_P. On the monadic layer, the MVars are also only allowed to have variables as content and therefore σ creates a binding, referenced by the MVar, that contains the real content. This is the same approach as used for the expression layer and also does not affect the runtime.

srnrp_A takes the minimal value w.r.t. the number of applications of (lbeta), (case) and (seq), this is also the case for mln_P, where the number of (Subst), (Branch) and (Seq) are considered. Hence we only need to compare the paths in the state tree with the corresponding paths of reduction sequences in CHF* using the parallel evaluation step.

For an easier analysis of differences in evaluation and therefore runtime, we put the abstract machine states into relation with the corresponding CHF*-processes. Since we compare the paths, each path is a sequence of CIOM1$_{sp,int}$-states.

CHF*-bindings are represented as usual heap-bindings in $\mathcal{H}$, MVars are aggregated in $\mathcal{M}$ and threads are in $\mathcal{T}$. Hence the structure of CHF*-processes differs from CIOM1$_{sp,int}$-states, but the runtime is identical. Thus we only need to consider the differences w.r.t. evaluation.

For most of the functional and monadic transformations there is no difference. The only difference occurs for the CHF*-rules (cpcxa) and (cpcxb): (cpcxa) is used to allow sharing of constructor applications. This is already achieved by σ and obviously does not affect the runtime. (cpcxb) also occurs on the transition from one CIOM1$_{sp,int}$-state to the next one, since for simplified expressions all constructor applications only have variables as arguments and therefore in this situation a (Lookup) followed by an immediate (Update) is triggered and leads to the same runtime.

In summary this shows the claim.

The complexity of a single transition step of CIOM1$_{t,par}$ increases practically, since a few more transition rules are allowed to be delayed by *DelTimeOpt* compared to *DelSpaceOpt*. However the asymptotical complexity is the same as for CIOM1$_{sp,par}$ as given by Proposition 6.9.

Note that the measure $\text{mln}_{P,C}$ can be used for correct runtime measurements for the abstract machines CIOM1$_{sp,int}$ and CIOM1$_{sp,par}$. Since both CIOM1$_{sp,int}$ and CIOM1$_{sp,par}$ calculate space optimal leafs of the complete state tree, that are not guaranteed to also have an optimal runtime, hence mln_P cannot be used for these two abstract machines.

6.3.4 Analysis-Tool CHFi

The abstract machines CIOM1$_{sp,par}$ and CIOM1$_{t,par}$ are both implemented in the analysis tool CHFi. Using this tool affirmative tests can be performed and also improvement-properties of transformations w.r.t. space or runtime can be falsified.

Moreover the algorithm SpOptN (see Algorithm 6.3) is implemented in CHFi using a buffer-approach: A calculation can be written to a specified buffer and in this way for independent calculations the input for SpOptN can be calculated directly by writing them in buffers. The call of SpOptN then only requires the buffers as input.

The tool also provides various features that can be handy in special situations, e.g. terminating as soon a leaf is discovered. For large state trees, where space-optimum or time-optimum cannot be calculated in appropriate time, this rather extreme kind of tree cutting yields an orientation. A template-system is provided to calculate a series of programs with a scalable input and also an LRP-mode.

The implemented measure interface can be used to extend the CHFi by arbitrary measures, where the only requirement is, that it is definite for two measures values if they are equal or one is greater. This interface allows extensions, e.g. the abstract machine CIOM1$_{t,par}$ is implemented only by defining a new measure.

The CHFi can be found at:

`http://www.ki.cs.uni-frankfurt.de/research/chfi`

6.3.5 Analyses of Examples

In the following subsections we perform analyses using the CHFi. Needed function definitions can be found in Section 2.6.

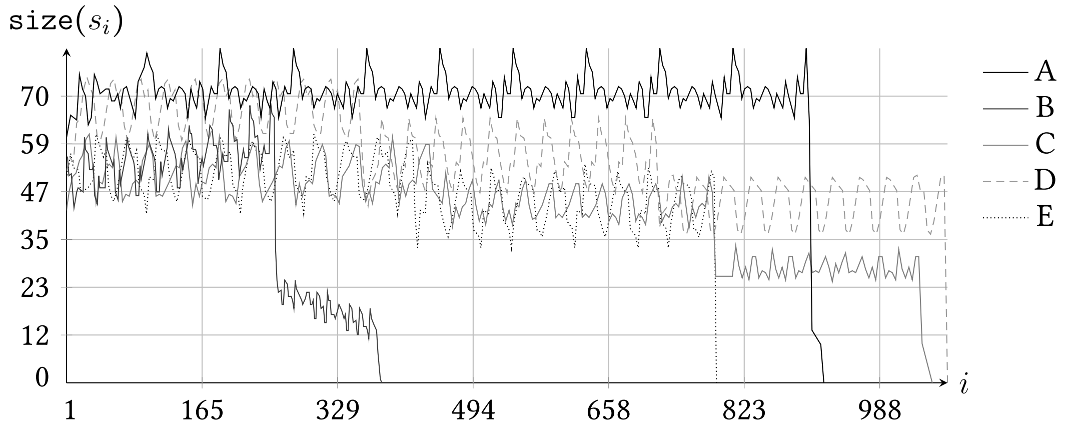
Required Space of Non-Interfering Processes

In this example we consider five programs named from A to E for simplicity. Program A applies `foldr` together with append on lists. Program B applies the linear reverse on a list (see Definition 3.21 for the definition of `reverse'`). Program C performs appends on lists. Program D is the same as the unshared example for `repeat` as defined in Section 3.6.2 for the analysis of sharing. Program E generates k-times lists containing k `True`-elements, checks whether all elements are `True`-elements and checks whether all of the elements of all of such lists are `True`-elements.

Program	Definition
A	`last (foldr (++) [] (take` k `(let` gen = `(False :` gen`) in` gen`)))`
B	`last (reverse' (take` k `(let` gen = `(True :` gen`) in` gen`)))`
C	`last (((replicate` k `True)++(replicate` k `True)) ++(replicate` k `True))`
D	`letrec` f = λy`.take` k `(repeat` y`) in and (allTrue (`f `True)) (and (allTrue (`f `True)) (allTrue (`f `True)))`
E	`allTrue (take` k `(repeat (allTrue (take` k `(repeat True)))))`

The CHFi is now used to generate the input for SpOptN (see Algorithm 6.3) as follows: The direct implementation of CIOM1$_{sp,par}$ only provides the calculated leaf, but we need the whole space trace. Therefore the CHFi is configured to calculate the whole state tree. For thread-free programs each of the state trees only consist of a single path and this applies to all of the five programs, hence delays can be turned off (note that still conflicts for MVars are considered, but there are no conflicts for independent threads). All of the resulting state trees are written to buffers. Then SpOptN can be called using the buffer numbers.

The following diagram shows all programs together for k = 10 where we see many overlaps. The result of SpOptN is 239.



Since all programs except of program B do not require to store intermediate data in memory, the optimal schedule can move the calculation of program B appropriately and therefore the required space only depends on the space consumption of B:

k	10	20	30	40	50	60	70	80	90	100
	spmin for specified program									
A	82	82	82	82	82	82	82	82	82	82
B	69	79	89	99	109	119	129	139	149	159
C	61	61	61	61	61	61	61	61	61	61
D	75	75	75	75	75	75	75	75	75	75
E	61	61	61	61	61	61	61	61	61	61
	Required space of all programs									
spmin	239	239	239	239	239	239	239	239	239	239

Also for all combinations, where certain processes have a higher k compared to the others, that have a fixed k of 10, the required space only depends on B. The reason for this is, that the schedule can be scaled, since only program B has a non-constant space behavior w.r.t. k. For further analyses we define two additional programs:

Program	Definition
F	`foldl and True (concat (take` k `(repeat (take 3 (repeat True)))))`
G	`foldr padd 0 (take` k `(repeat 3))`

The program F has a space consumption that behaves quadratic since `foldl` is used (see Section 3.6.2 for more details) and program G has a linear space consumption. Note that each Peano number is counted as size 1, hence there is no difference in space consumption if a high number is replaced by a low number. In the following table the results are summarized for different combinations, where a subset of the three considered programs is analyzed.

k	10	20	30	40	50	60	70	80	90	100
	spmin for specified program									
B	69	79	89	99	109	119	129	139	149	159
F	242	482	722	962	1202	1442	1682	1922	2162	2402
G	118	168	218	268	318	368	418	468	518	568
	Required space, if k used for specified programs, all others use $k = 10$									
B	269	269	269	269	269	269	269	269	269	269
F	269	509	749	989	1229	1469	1709	1949	2189	2429
G	269	286	286	286	320	370	420	470	520	570
B, F	269	509	749	962	1229	1469	1709	1949	2189	2429
B, G	269	286	286	286	320	370	420	470	520	570
F, G	269	526	766	1006	1246	1486	1726	1966	2206	2446
B, F, G	269	526	766	1006	1246	1486	1726	1966	2206	2446

Program B does not have an impact on the required space, since it is always dominated by program F and even an optimal schedule as calculated by SpOptN here for all cases cannot improve the space consumption by a smart rearrangement. This is clearly visible in the case where only B has input size k and also if we compare the case where

F and G has input size k with the case where all processes have input size k. The same argumentation applies to the case where B and G have input size k compared to the case where only G has input size k.

In the case, where G has input size k, the required space increases linearly, but not for low input sizes. For low input sizes the scheduler can rearrange the threads so that the required space remains at 286 or lower.

As expected, in all cases where F is involved, the required space increases drastically.

Parallelization of List-Evaluation

If a long is constructed and evaluated afterwards, then it seems to be a good idea to parallelize the construction of the list if possible. Consider the following programs:

```
s := let res = foldr padd 0 (replicate (2·k) 1)
       in seq res (return res)
t := do n1 ⇐ future (return (let res = foldr padd 0 (replicate k 1)
                                  in seq res res))
        n2 ⇐ future (return (let res = foldr padd 0 (replicate k 1)
                                  in seq res res))
        let res = padd n1 n2
        seq res (return res)
```

The pure program s constructs a list containing $2 \cdot k$-times the number 1 and then uses `foldr` to calculate the sum of all of these numbers. t is the same as s but parallelizes the list-constructions and most additions.

Since we are interested in an improvement of runtime and want to analyze the space consumption, delays are turned off. For space analyses $\text{mln}_{P,A}$ denotes the mln_P-value of the space-optimal path, hence $\text{mln}_{P,A}$ is an approximation of mln_P.

Then we get the following results:

k	10	20	30	40	50	60	70	80	90	100
	Pure variant (s)									
mln_P	208	408	608	808	1008	1208	1408	1608	1808	2008
$\text{mln}_{P,A}$	208	408	608	808	1008	1208	1408	1608	1808	2008
spmin	158	258	358	458	558	658	758	858	958	1058
	Parallel variant (t)									
mln_P	114	214	314	414	514	614	714	814	914	1014
$\text{mln}_{P,A}$	114	214	314	414	514	614	714	814	914	1014
spmin	122	172	222	272	322	372	422	472	522	572

The parallelization nearly halves both runtime and space consumption and since all $\text{mln}_{P,A}$-values are the same as the corresponding mln_P-values, the required *spmin*-values correspond to optimal runtime-values. The reason for the space decrease is that the whole list can be garbage collected from left to right and using two threads, the garbage collector can remove two elements in the parallel variant in contrast to one element in the pure variant.

The table above is generated as follows:

The doubling of k for the pure variant is directly achieved using the flexible template system. Two calculations are performed separately. The first calculation yields *spmin* and the corresponding $\mathrm{mln}_{P,A}$-value and the second one only calculates the mln_P-values.

For both calculations no delays are needed, since the space consumption is intended to be improved while keeping a good runtime. This approach is correct, since there are no conflicts between threads and the results of all threads are needed. Moreover for both cases the full state tree mode is used, since there are no dependencies between the threads, hence there is only a path and the calculation of the state tree has the least overhead.

For the second case the garbage collector is turned off completely. Note that this might increase the memory consumption of the CHFi, hence in some comparable cases it might be better to configure the garbage collector to be active with a lower frequency instead of turning it off.

Fold using Addition on Binary Tree

In this analysis an introductionary improvement-example of (SSSD18) w.r.t. runtime is considered and we also want to confirm the improvement-property w.r.t. space consumption. The program is slightly adapted as follows:

```
s := let calcPure = (λn.case n of {
                        ((Leaf k) → k)
                        ((Node l r) → padd (calcPure l) (calcPure r)) })
         res = calcPure tree
     in seq res (return res)
t := let calcFut = (λn.case n of {
                       ((Leaf k) → return k)
                       ((Node l r) → do lres ⇐ future (calcFut l)
                                        rres ⇐ future (calcFut r)
                                        let res = padd lres rres
                                        seq res (return res)) })
     in calcFut tree
```

We assume that the input $tree$ is a binary tree that already is fully evaluated and all leafs contain the value 0. The zeroes allow an easier reasoning, since no laziness w.r.t. Peano addition is introduced by zeroes. Both programs fold the addition operator over the whole tree, but s is a pure implementation while t parallelizes as much as possible, if many of the threads are able to run in parallel.

Since the impact of parallelization on space consumption is the goal of this analysis, the CHFi is configured not to use any delays. This is correct, since there are no conflicts between the threads and the results of all threads are needed.

Two calculations are needed. The target of the first calculation is the space consumption and mln_P-value of the space-optimal path, denoted as $\mathrm{mln}_{P,A}$. The target of the second calculation is the optimal overall runtime mln_P, where the garbage collector can be turned off for a shorter calculation-time.

k	10	20	30	40	50	60	70	80	90	100
	Pure variant (s)									
mln_P	10	24	52	108	220	444	892	1788	3580	7164
$\mathrm{mln}_{P,A}$	10	24	52	108	220	444	892	1788	3580	7164
spmin	36	47	64	93	146	247	444	833	1606	3147
	Parallel variant (t)									
mln_P	12	28	52	88	137	218	349	626	1133	2165
$\mathrm{mln}_{P,A}$	12	28	52	88	137	218	349	626	1133	2165
spmin	54	84	144	253	437	724	1263	2336	4424	8566

The parallelization has a great impact on the runtime (about 70 percent for $k = 100$) but simultaneously the space consumption increases by a multiplicative constant (about 2.7 for $k = 100$).

The pure variant needs less space, since the calculation proceeds over the whole tree using a depth-first-order and therefore all intermediate calculations can be performed locally, using as least memory as possible. The parallel variant calculates as much as possible in parallel and thus the space requirement of all additions sums up quickly. As expected the $\mathrm{mln}_{P,A}$ and mln_P-values coincide, hence the calculated *spmin*-values always coincide with optimal runtime-values.

Thus if a great amount of memory is available, then the parallelization can be applied to improve the runtime noticeably. If the memory is limited, then it might be better to use the pure variant. However in practice the number of processors is usually a restriction, hence the improvement of runtime can be expected to be weaker in practice than in this analysis and also the space increase is consequently not that high.

Common Subexpression Elimination

The example used in the proof of Proposition 3.7 to show that common subexpression elimination (cse) is not a space improvement in LRP can be directly transferred to CHF* and a short test using CHFi affirmed, that common subexpression elimination is not a space improvement. However in this example the improvement of runtime using (cse) is only an additive constant and therefore we now analyze a case with higher impact on runtime.

The following programs s and t both use two threads that each calculate a Boolean value and in the end the logical and is applied on the two Boolean values. The calculation of the Boolean values are based on the same list of Boolean values, where s performs the generation of those lists separately in the calculations of both threads, while t uses an extra thread, that calculates the needed list. Hence t is s after some general form of common subexpression elimination is applied.

```
s := do b1 ⇐ future (return (last (replicate k True)))
        b2 ⇐ future (return (allTrue (replicate k True)))
        let res = and b1 b2
        seq res (return res)
t := do lst ⇐ future (return (replicate k True)
        b1  ⇐ future (return (last lst))
        b2  ⇐ future (return (allTrue lst))
        let res = and b1 b2
        seq res (return res)
```

Now the CHFi is configured to use an iterative depth-first-search using checksums. α-equivalence-testing is turned off, but the delay-function *DelSpaceOpt* is used, since the target is the required space. This calculation leads to the following *spmin*-values with the corresponding runtime values $\text{mln}_{P,A}$.

A second calculation yields the optimal overall runtime mln_P, where the garbage collector can be turned off for a shorter calculation-time. Also no delays are needed, since there are no conflicts between the threads and moreover it is known that the results of all threads are required for the final result.

k	5	10	15	20	25	30	35
	Pure variant (s)						
mln_P	74	134	194	254	314	374	434
$\text{mln}_{P,A}$	74	134	194	254	314	374	434
spmin	56	56	56	56	56	56	56
	Parallel variant (t)						
mln_P	57	102	147	192	237	282	327
$\text{mln}_{P,A}$	57	102	147	192	237	282	327
spmin	55	60	65	70	75	84	94

The space consumption for s is constant, since `last` and `allTrue` work on separated lists and therefore the list can be garbage collected from left to right. For t the space consumption is linear in the length of the list, since the future lst implements a sharing that forces the complete calculation of the list. For both programs the runtime is linear, however the runtime improves by at most 25 percent if common subexpression is used.

Optimizing Factory and Limitations

In the following example a small factory is considered. The goal is to model the space used in the factory building. The idea is to analyze the effectiveness of buying additional machines.

It is assumed that only one product is produced in the factory. The factory has six machines $m_1, \ldots, m_6$ in total, producing different subproducts, where some machines require already produced subproducts of other machines. The machines together with the subproducts they produce and the requirements are defined by the following table:

Machine	Subproduct	Required Subproducts (Amounts)
m_1	sp_1	
m_2	sp_2	
m_3	sp_3	
m_4	sp_4	sp_1 (2), sp_2 (1)
m_5	sp_5	sp_2 (1), sp_3 (2)
m_6	sp_6 (final product)	sp_4 (1), sp_5 (1)

The factory can be visualized as follows, where the lines represent conveyors and the numbers at each arrow states how many subproducts of the machine, where the arrow starts at, are needed for the machine or point where the arrow ends:

$$\begin{tikzcd}
m_1 \arrow[d, "2"'] & m_2 \arrow[d, "2"'] & m_3 \arrow[d, "2"] \\
m_4 \arrow[dr, "1"'] & {} \arrow[l, "1"'] \arrow[r, "1"] & m_5 \arrow[dl, "1"] \\
& m_6 &
\end{tikzcd}$$

The machines $m_1, \ldots, m_3$ can be modeled in CHF* using three MVars and three threads. For each machine exists an MVar, say $mv_1, \ldots, mv_3$, that contains a list of corresponding subproducts, e.g. $[sp_1, sp_1, \ldots]$ for m_1. For each machine $m_1, \ldots, m_3$ a thread exists that runs in an infinite loop, extending the list of the corresponding MVar by finished subproducts. Since the termination of the main-thread terminates all other threads, this does not affect the overall termination.

Then the machines m_4 and m_5 take the needed subproducts (respecting the requirements as defined by the table above) of the MVars $mv1, \ldots, mv3$ and write back the reduced lists to the corresponding MVars. If the needed amount of subproducts is not reached, then the affected list is not changed and written back to the MVar. The same approach as for m_4 and m_5 can be used to model m_6. Since the lists are empty at the start of the program and we are looking for a minimum of space consumption, it is necessary to require that m_6 produces a certain amount of products (e.g. the amount of products per day). To get an intuition of the interpretation of space consumption, consider the following factory layout, where m_2 is duplicated:

$$\begin{tikzcd}
m_1 \arrow[d, "2"'] & m_2 \arrow[dl, "1"] & m_2' \arrow[dr, "1"'] & m_3 \arrow[d, "2"] \\
m_4 \arrow[dr, "1"'] & & & m_5 \arrow[dll, "1"] \\
& m_6 & &
\end{tikzcd}$$

The space consumption implicitly takes the needed real space in the factory building of each machine into account and also the length of conveyors can be reduced, if the minimal space consumption is known. Adjusting the certain amount of products of m_6 simulates the flow and is interesting if new machines are added, hence to compare the both factory layouts above.

The machine m_1 can be implemented as follows (m_2, m_2' and m_3 can be implemented similarly), where prodSP1 is a function that simulates the production of subproduct sp_1 by m_1:

```
do sp1 ⇐ newMVar Nil
   t1 ⇐ future (letrec inf = do lst ⇐ takeMVar sp1)
                                putMVar sp1 (prodSP1 : lst)
                in inf)
```

The implementation of machine m_4 is straightforward, however it needs a few more operations (m_5 can be implemented similarly), where > as in Haskell can be implemented using a case-expression and all other Haskell-constructs (e.g. lst !! k returns the k-th element of lst) are also implemented using case-expressions. Also prodSP4 takes two subproducts sp_1 and one subproduct sp_2 and simulates the production of one subproduct sp_4.

```
do sp4 ⇐ newMVar Nil
   t4 ⇐ future (do sp1lst ⇐ takeMVar sp1
                   sp2lst ⇐ takeMVar sp2
                   sp4lst ⇐ takeMVar sp4
                   if (length sp1lst > 1) && (length sp2lst > 0)
                      then do putMVar sp1 (tail (tail sp1lst))
                              putMVar sp2 (tail sp2lst)
                              putMVar sp4 ((prodSP4
                                                (head sp1lst)
                                                (head (tail sp1lst))
                                                (head sp2lst))
                                                  : sp4lst)
                      else do putMVar sp1 sp1lst
                              putMVar sp2 sp2lst
                              putMVar sp4 sp4lst)
```

It is correct to write $sp1lst$ and $sp4lst$ back to the corresponding MVars, since all threads require a filled MVar to proceed, hence the thread of machine m_4 blocks all affected MVars. Note that also m_6 can be implemented as above for m_4, however some additional program logic is needed to implement that a certain number of products should be produced.

This whole example shows the computation limit and can be used to analyze the CHFi using different configurations.

First of all the checksum-testing and especially α-equivalence-testing of states can be helpful (e.g. in the example for common subexpression elimination above the checksums are effective), but this hardly depends on the considered program. If delays lead to a high amount of nearly identical states, then the chance to get rid of big subtrees is high and in these cases the extra runtime needed by α-equivalence-testing also help to cut off subtrees. However there never is a guarantee that these optimizations reduce the overall calculation time.

The initial maximal depth and the amount of increase, if the maximal depth needs to be enhanced, have a great impact. For cases where leafs with low space consumption are detected with a low depth, the initial maximal depth should be quite low. But there are cases, where the first leaf is found with a high depth and then the initial maximal depth needs to be set a lot higher.

Moreover the leftmost path might be the longest and then the calculation time might increase a lot, especially if the shortest path is the rightmost one. We consider the factory example above and assume that only the machines m_1 to m_3 are implemented and then let the main-thread finishes its calculation with a take-operation on the MVar $sp1$. The shortest calculation is clearly, that the main-thread performs its take-operation and the whole calculation terminates, while all other threads pause. But the depth-first-search takes the longest path first, while the optimal path is on the right side of the tree, hence the optimal path is only found if the maximal depth is reached. But even if this is the case, the optimal path is also long because of the amount of reduction steps caused by the complexity of the program, hence effectively a big part of the tree needs to be processed before the optimal path is found. Permuting the order in which the children are processed might only help in this explicit scenario, but in general the optimal path can go through the whole tree without visible order of the next elements seen locally from a single node.

If we look at the implementation of m_4 above, then the program is clearly too large w.r.t. minimal runtime itself and also the number of threads. We give a strategy for such cases, that however yields no results for the factory example, since the state tree is too large:

1. Check whether only iterative depth-first-search or using checksums or also using α-equivalence is the best.
2. Try different initial maximal depth configurations and also adapt the constant that controls the increase of maximal depth.
3. The full tree evaluation provides a tree cut, as soon as a leaf is found. This way an approximation is calculated the helps to get an orientation.
4. If there are no conflicts between threads, then delays can be turned off.
   - For space analyses this leads to an approximation.
   - For time analyses it depends, if it is known, that all results from the threads are fully needed (hence no thread performs an unnecessary reduction step). If this is the case, then turning off delays yields the optimal runtime with a much lower calculation time. If this cannot be guaranteed, then the result is an approximation.

7 Conclusion and Future Work

In this thesis the space consumption and runtime is analyzed using the calculi LRP and CHF. Both calculi are slightly adapted for an easier space analysis, yielding LRPgc and CHF*, where all modifications preserve the semantics. This work focuses primarily on the minimization of the maximal space consumption during execution.

The notions of space improvement and space equivalence are introduced in different variants for LRPgc and LR: Using an efficiently implementable eager garbage collector that is applied whenever possible or using an eager total garbage collector that replaces any subexpressions by a non-terminating constant with size zero, if the overall termination is not affected. Also context lemmas are proved for (total) space improvement and equivalences. The context lemmas are used to show the (total) space improvement or equivalence properties of several transformations.

Compared to (GS99, GS01) the space measurement and definitions of space improvements used for LRPgc turns out to be stable and effective. Also examples of (GS01) can be analyzed easier using our method.

The compatibility between LRPgc and the introduced abstract machine $M1_{sp}$ leads to analyses of the space consumption and runtime of more complex programs using an implementation of $M1_{sp}$.

We also develop an efficient algorithm to calculate the required space w.r.t. schedules of multiple independent processes, where only start- and end-points are required to be in sync. Using Boolean conditions on simultaneous or relative time points of two processes, the required space can be calculated in exponential time, however the time is still polynomial, if the number of processors and conditions is fixed. For the general case of synchronizations, that is NP-complete, the abstract machines $CIOM1_{sp,par}$ and $CIOM1_{t,par}$ are introduced and help to find the required space or minimal runtime.

The abstract machines $CIOM1_{sp,par}$ and $CIOM1_{t,par}$ intuitively both build up a tree of all needed possibilities to find an optimal solution. The tree can be large caused by nondeterminism and the different interleavings of threads that affect the overall space consumption. Optimizations of both machines are tree cuts, where subtrees are removed, that not contribute to the optimal solution and also checksum- and α-equivalence-testing helps to eliminate semantic equal nodes in the tree.

Implementations of the machines can easily be extended and also different options, that effect the overall runtime, can be used. In any case the impact of each optimization heavily depends on the given input. Despite of the worst-case complexity of the tree, we performed several analyses successfully.

As future work different space measures can be considered. The maximal space consumption during the execution of a program is optimized in this work and this applies to various scenarios. However sometimes the average space consumption is interesting and then the used space measure might be not appropriate.

Moreover if we know that a space increase is temporary in any case, then we also might accept this little space increase. One approach is a measure that abstracts over such temporary increases, another approach would be to glue some sequences of transformations together. The last approach can be implemented using forking diagrams.

If we compare an optimal garbage collector with total garbage collection, then it seems that the difference is negligible. The relationship between the implementable eager garbage collection as used by LRPgc and total garbage collection is a topic for future research. Also different garbage collection approaches can be used in the implementations, where (Wil92) shows different basic techniques.

The notions of space improvement and equivalence can be transferred to CHF*. The approach for runtime improvements in (SSSD18) can be utilized: A context lemma is not needed, instead normalized reduction sequences can be used and forking diagrams need to be calculated directly, where care has to be taken w.r.t. garbage collection. Using this approach it seems viable to show the space improvement or equivalence properties of several transformations.

The abstract machines $\text{CIOM1}_{sp,par}$ and $\text{CIOM1}_{t,par}$ can be optimized further, e.g. for some specific (Update)-transitions it is clear, that a delay of the corresponding thread cannot improve the space consumption and thus a complete subtree can be removed. However such optimizations are sophisticated since the check itself is required to be very quick, otherwise the overall optimization of runtime is insignificant.

Bibliography

[Abr90] Samson Abramsky. *The Lazy Lambda Calculus*, page 65–116. Addison-Wesley Longman Publishing Co., Inc., USA, 1990.

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces.* Arpaci-Dusseau Books, 1.00 edition, August 2018.

[ADN08] Christian Artigues, Sophie Demassey, and Emmanuel Néro. *Resource-Constrained Project Scheduling.* ISTE and Wiley, London, UK, and Hoboken, USA, 2008.

[AFM$^+$95] Zena. M. Ariola, Matthias Felleisen, John. Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *POPL'95*, pages 233–246, San Francisco, California, 1995. ACM Press.

[BFKT00] Clem Baker-Finch, David J. King, and Phil Trinder. An operational semantics for parallel lazy evaluation. *SIGPLAN Not.*, 35:162–173, 2000.

[BMV13] Andreas Beckmann, Ulrich Meyer, and David Veith. An implementation of i/o-efficient dynamic breadth-first search using level-aligned hierarchical clustering. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8125 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 2013.

[BR00] Adam Bakewell and Colin Runciman. A model for comparing the space usage of lazy evaluators. In *PPDP*, pages 151–162, 2000.

[CLPJ$^+$07] Manuel Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: A status report. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 10–18, 01 2007.

[Dal16] Nils Dallmeyer. *Design and implementation of a test suite for exploring space improvements in a call-by-need functional language with polymorphic types.* Msc. thesis, Institut für Informatik, Goethe-Universität Frankfurt am Main, 2016.

[DCL18] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on nosql stores. *ACM Comput. Surv.*, 51(2), April 2018.

[DS16] Nils Dallmeyer and Manfred Schmidt-Schauß. An environment for analyzing space optimizations in call-by-need functional languages. In Horatiu Cirstea and Santiago Escobar, editors, *Proc. 3rd WPTE@FSCD*, volume 235 of *EPTCS*, pages 78–92, 2016.

[FF99] Cormac Flanagan and Mattias Felleisen. The semantics of future and an application. *J. Funct. Programming*, 9:1–31, 1999.

[Gar76] M. R. Garey. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[GJ77] M. R. Garey and D. S. Johnson. Two-processor scheduling with start-times and deadlines. *Siam J. Comput.*, 6(3):316–426, 1977.

[GLJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[GS99] Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. *ENTCS*, 26:69–86, 1999.

[GS01] Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In Benjamin C. Pierce, editor, *Proc Sixth ACM (ICFP '01), Firenze, Italy*, pages 265–276, 2001.

[Gus01] Jörgen Gustavsson. *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2001.

[HH14] Jennifer Hackett and Graham Hutton. Worker/wrapper/makes it/faster. In *ICFP '14*, pages 95–107, 2014.

[HH19] Jennifer Hackett and Graham Hutton. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[HJ11] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *2011 International Conference on Cloud and Service Computing*, pages 336–341, Dec 2011.

[HMP+18] Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. I/o-efficient generation of massive graphs following the lfr benchmark. *Journal of Experimental Algorithmics*, 23:1–33, 08 2018.

[HMPJ05] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell'05 - Proceedings of the ACM SIGPLAN 2005 Haskell Workshop*, pages 49–61, 01 2005.

[Hug89] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[JV05] Patricia Johann and Janis Voigtländer. The Impact of Seq on Free Theorems-Based Program Transformations. *Fundam. Inf.*, 69(1-2):63–102, July 2005.

[LM99] Søren B. Lassen and Andrew Moran. Unique fixed point induction for McCarthy's Amb. In Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Proc. 24th MFCS'99*, volume 1672 of *LNCS*, pages 198–208. Springer, 1999.

[LOMPnM05] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña Marí. Parallel functional programming in Eden. *J. Funct. Programming*, 15:431–475, 2005.

[LRS$^+$03] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G.J. Michaelson, R. Peña, S. Priebe, Á.J. Rebón, and P.W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, Sep 2003.

[MBCP14] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 325–337, New York, NY, USA, 2014. Association for Computing Machinery.

[MNJ11] Simon Marlow, Ryan Newton, and Simon L. Peyton Jones. A monad for deterministic parallelism. In Koen Claessen, editor, *Proc. 4th Haskell Symposium 2011*, pages 71–82. ACM, 2011.

[MS99] 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*, POPL '99, pages 43–56, New York, NY, USA, 1999. ACM.

[MSC99] Andrew K. D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Comput. Sci.*, pages 85–102. Springer-Verlag, 1999.

[MSV12] Loris Marchal, Oliver Sinnen, and Frédéric Vivien. Scheduling tree-shaped task graphs to minimize memory and makespan. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, 10 2012.

[NSS06] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoret. Comput. Sci.*, 364(3):338–356, 2006.

[NSSSS07] Joachim Niehren, David Sabel, Manfred Schmidt-Schauß, and Jan Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337, 2007.

[Pey01] Simon L. Peyton Jones. Tackling the awkward squad: monadic input/-output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS-Press, 2001.

[PGF96] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proc. 23rd ACM POPL 1996*, pages 295–308. ACM, 1996.

[PJ87] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, January 1987.

[PS09] Simon L. Peyton Jones and Satnam Singh. A tutorial on parallel and concurrent programming in Haskell. In *Advanced Functional Programming, 6th International School, Revised Lectures*, volume 5832 of *Lecture Notes in Comput. Sci.*, pages 267–305. Springer, 2009.

[PW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proc. 20th ACM POPL 1993*, pages 71–84. ACM, 1993.

[RMZ12] Juan Manuel Rodríguez, Cristian Mateos, and Alejandro Zunino. Are smartphones really useful for scientific computing? In Francisco Cipolla-Ficarra, Kim Veltman, Domen Verber, Miguel Cipolla-Ficarra, and Florian Kammüller, editors, *Advances in New Technologies, Interactive Interfaces and Communicability*, pages 38–47, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Sab12] David Sabel. An abstract machine for Concurrent Haskell with futures. In Stefan Jähnichen, Bernhard Rumpe, and Holger Schlingloff, editors, *Software Engineering 2012 Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar - 2. März 2012 in Berlin*, volume 199 of *GI Edition - Lecture Notes in Informatics*, pages 29–44, February 2012. (5. Arbeitstagung Programmiersprachen (ATPS'12)).

[San91] David Sands. Operational theories of improvement in functional languages (extended abstract). In Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler, editors, *Proc. 1991 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 298–311. Springer, 1991.

[San95a] David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5:495–541, 1995.

[San95b] David Sands. Total Correctness by Local Improvement in Program Transformation. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 221–232, New York, NY, USA, 1995. ACM.

[San97] David Sands. From SOS rules to proof principles: An operational metatheory for functional languages. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Proc. 24th ACM POPL 1997*, pages 428–441. ACM Press, 1997.

[San98] Peter Sanders. Random permutations on distributed, external and hierarchical memory. *Inf. Process. Lett.*, 67:305–309, 1998.

[Ses97] Peter Sestoft. Deriving a Lazy Abstract Machine. *J. Funct. Program.*, 7(3):231–264, May 1997.

[SS07] Manfred Schmidt-Schauß. Correctness of copy in calculi with letrec. In *Term Rewriting and Applications*, Lecture Notes in Comput. Sci., pages 329–343. Springer, 2007.

[SS11] David Sabel and Manfred Schmidt-Schauß. A contextual semantics for concurrent Haskell with futures. In Peter Schneider-Kamp and Michael Hanus, editors, *Proc. 13th ACM PPDP 2011*, pages 101–112. ACM, 2011.

[SS14] Manfred Schmidt-Schauß and David Sabel. Contextual equivalences in call-by-need and call-by-name polymorphically typed calculi (preliminary report). In Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba, editors, *WPTE 2014*, volume 40 of *OASICS*, pages 63–74. Schloss Dagstuhl, 2014.

[SS15] Manfred Schmidt-Schauß and David Sabel. Improvements in a functional core language with call-by-need operational semantics. In Moreno Falaschi and Elvira Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 220–231. ACM, July 2015.

[SSD17] Manfred Schmidt-Schauß and Nils Dallmeyer. Space improvements and equivalences in a polymorphically typed functional core language: Context lemmas and proofs. Frank report 57, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, July 2017.

[SSD18] Manfred Schmidt-Schauß and Nils Dallmeyer. Space improvements and equivalences in a functional core language. In Horatiu Cirstea and David Sabel, editors, Proceedings Fourth International Workshop on *Rewriting Techniques for Program Transformations and Evaluation,* Oxford, UK, 8th September 2017, volume 265 of *Electronic Proceedings in Theoretical Computer Science*, pages 98–112. Open Publishing Association, 2018.

[SSD19a] Manfred Schmidt-Schauß and Nils Dallmeyer. Optimizing space of parallel processes. In Joachim Niehren and David Sabel, editors, Proceedings Fifth International Workshop on *Rewriting Techniques for Program Transformations and Evaluation,* Oxford, UK, 8th July 2018, volume 289 of *Electronic Proceedings in Theoretical Computer Science*, pages 68–87. Open Publishing Association, 2019.

[SSD19b] Manfred Schmidt-Schauß and Nils Dallmeyer. Space improvements for total garbage collection. Frank report 61, Institut für Informatik, Goethe-Universität Frankfurt am Main, April 2019.

[SSS08a] David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

[SSS08b] Manfred Schmidt-Schauß, David Sabel, and Marko Schütz. Safety of Nöcker's Strictness Analysis. *J. Funct. Program.*, 18(4):503–551, July 2008.

[SSS10] Manfred Schmidt-Schauß and David Sabel. Closures of may-, should- and must-convergences for contextual equivalence. *Inf. Process. Lett.*, 110:232–235, 02 2010.

[SSS12] David Sabel and Manfred Schmidt-Schauß. Conservative concurrency in Haskell. In Nachum Dershowitz, editor, *Proc. 27th IEEE LICS 2012*, pages 561–570. IEEE, 2012.

[SSS15a] Manfred Schmidt-Schauß and David Sabel. Sharing-aware improvements in a call-by-need functional core language. In Ralf Lämmel, editor, *Proc. 27th IFL 2015*, pages 6:1–6:12, New York, NY, USA, 2015. ACM.

[SSS15b] Manfred Schmidt-Schauß and David Sabel. Sharing decorations for improvements in a functional core language with call-by-need operational semantics. Frank report 56, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, September 2015.

[SSS16a] David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda calculus with scoped work decorations. In Wolf Zimmermann and et.al., editors, *Software Engineering Workshops 2016*, volume 1559 of *CEUR Workshop Proceedings*, pages 70–90. CEUR-WS.org, 2016.

[SSS16b] Manfred Schmidt-Schauß and David Sabel. Improvements in a functional core language with call-by-need operational semantics. Frank report 55, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, August 2016.

[SSS17] Manfred Schmidt-Schauß and David Sabel. Improvements in a call-by-need functional core language: Common subexpression elimination and resource preserving translations. *Science of Computer Programming*, 147:3–26, 2017.

[SSSD18] Manfred Schmidt-Schauß, David Sabel, and Nils Dallmeyer. Sequential and parallel improvements in a concurrent functional programming language. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, pages 20:1–20:13, New York, NY, USA, 2018. ACM.

[SSSH09] David Sabel, Manfred Schmidt-Schauß, and Frederik Harwath. Reasoning about contextual equivalence: From untyped to polymorphically typed calculi. In *INFORMATIK 2009,4. Arbeitstagung Programmiersprachen*, LNI 154, pages 369; 2931–45, 2009.

[Sve02] Josef Svenningsson. Shortcut Fusion for Accumulating Parameters & Zip-like Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 124–132, New York, NY, USA, 2002. ACM.

[THLP98] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Funct. Programming*, 8(1):23–60, 1998.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, Berlin, Heidelberg, 1995. Springer-Verlag.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, page 1–42, Berlin, Heidelberg, 1992. Springer.

Index

Symbols, Measures and Relations

Nils Dallmeyer

Lebenslauf

Persönliche Daten

Geburtstag	13.12.1991
Geburtsort	Hanau

Ausbildung und beruflicher Werdegang

2016–2020	**Promotionsstudium in der Informatik**, *Goethe-Universität*, Frankfurt.
2014–2016	**Master-Studium der Informatik**, *Goethe-Universität*, Frankfurt. Abschlussarbeit: *Design and Implementation of a Test Suite for Exploring Space Improvements in a Call-by-Need Functional Language with Polymorphic Types* (Betreuer: Prof. Dr. Schmidt-Schauß)
2011–2014	**Bachelor-Studium der Informatik**, *Goethe-Universität*, Frankfurt. Abschlussarbeit: *Implementierung einer interaktiven dreidimensionalen Anzeige von Bildausschnitten von Grammatik-komprimierten Bildern in der funktionalen Programmiersprache Haskell* (Betreuer: Prof. Dr. Schmidt-Schauß)
2002–2011	**Gymnasium**, *Hohe Landesschule*, Hanau. Abitur