

Runtime Repeated Recursion Unfolding in CHR: A Just-In-Time Online Program Optimization Strategy That Can Achieve Super-Linear Speedup

Thom Frühwirth

University of Ulm, Germany
thom.fruehwirth@uni-ulm.de

April 25, 2024 minor revision of major revision of submission

Abstract. We introduce a just-in-time runtime program transformation strategy based on repeated recursion unfolding. Our online program optimization generates several versions of a recursion differentiated by the minimal number of recursive steps covered. The base case of the recursion is ignored in our technique.

Our method is introduced here on the basis of single linear direct recursive rules. When a recursive call is encountered at runtime, first an unfolder creates specializations of the associated recursive rule on-the-fly and then an interpreter applies these rules to the call. Our approach reduces the number of recursive rule applications to its logarithm at the expense of introducing a logarithmic number of generic unfolded rules.

We prove correctness of our online optimization technique and determine its time complexity. For recursions which have enough simplifiable unfoldings, a super-linear is possible, i.e. speedup by more than a constant factor. The necessary simplification is problem-specific and has to be provided at compile-time. In our speedup analysis, we prove a sufficient condition as well as a sufficient and necessary condition for super-linear speedup relating the complexity of the recursive steps of the original rule and the unfolded rules.

We have implemented an unfolder and meta-interpreter for runtime repeated recursion unfolding with just five rules in Constraint Handling Rules (CHR) embedded in Prolog. We illustrate the feasibility of our approach with simplifications, time complexity results and benchmarks for some basic tractable algorithms. The simplifications require some insight and were derived manually. The runtime improvement quickly reaches several orders of magnitude, consistent with the super-linear speedup predicted by our theorems.

Keywords. Just-In-Time Program Transformation, Runtime Program Optimization, Online Program Specialization, Repeated Recursion Unfolding, Super-Linear Speedup, Recursion, Meta-Interpreter, Speedup Theorem, Time Complexity.

Contents

1	Introduction	3
2	Preliminaries	4
2.1	Abstract Syntax of CHR	4
2.2	Abstract Operational Semantics of CHR	5
3	Runtime Repeated Recursion Unfolding	7
3.1	Rule Unfolding	7
3.2	Rule Simplification	9
3.3	Runtime Repeated Recursion Unfolding	10
3.4	Optimal Rule Applications	14
4	Implementation of Runtime Repeated Recursion Unfolding	15
4.1	Unfolder Implementation	16
4.2	Meta-Interpreter Implementation	18
4.3	Recursive Constraint Implementation	19
5	Time Complexity of the Implementation	20
5.1	Time Complexity of the Original Rule	20
5.2	Time Complexity of the Unfolder	20
5.3	Time Complexity of the Meta-Interpreter	21
5.4	Time Complexity of Runtime Repeated Recursion Unfolding	22
6	Super-Linear Speedup Theorems	23
6.1	Solving the Recurrences for Polylog-Polynomial Time Complexities	23
6.2	Sufficient Condition for Super-Linear Speedup	25
6.3	Sufficient and Necessary Condition for Super-Linear Speedup	26
7	Experimental Evaluation: Examples with Benchmarks	27
7.1	Summation Example, Contd.	27
7.2	List Reversal Example	31
7.3	Sorting Example	35
8	Related Work	40
9	Discussion	41
10	Conclusions and Future Work	43

The table of contents is included for reviewing purposes only

1. Introduction

In the context of rule-based programming, *unfolding* is a program transformation that basically replaces a call in the body (right-hand side) of a rule with the body of a rule whose head (left-hand side) is matched by the call. *Repeated recursion unfolding* [1] first unfolds a given recursive rule with itself and simplifies it. This results in a specialized recursive rule that covers two recursive steps instead of one. It continues to unfold the last unfolded recursive rule with itself. Each unfolding doubles the number of recursive steps covered by the unfolded rule. In this article, we extend the method to an on-line program optimization and give an implementation of the necessary unfolders and interpreter. The given call determines how far the unfolding proceeds and how many rules are generated. Therefore the optimization cannot be performed at compile-time.

Example 1.1. (Summation)

Consider the following toy example, a simple recursive program written in abstract syntax of the programming language Constraint Handling Rules (CHR). It recursively adds all numbers from 1 to n . The rules can be understood as a procedure definition for a binary relation *sum*. Rule b covers the base case and rule r covers the recursive case.

$$\begin{aligned} b : \text{sum}(N, S) &\Leftrightarrow N = 1 \mid S = 1 \\ r : \text{sum}(N, S) &\Leftrightarrow N > 1 \mid \text{sum}(N-1, S1), S = N+S1 \end{aligned}$$

The head $\text{sum}(N, S)$, guard (e.g. $N = 1$) and body of a rule are separated by the symbols $\Leftrightarrow$ and $\mid$, respectively. Upper case letters stand for variables. When a call matches the head of a rule and the guard condition holds, the body of the rule is executed.

Unfolding the recursive rule with a copy of itself and simplifying the resulting rule gives

$$r_1 : \text{sum}(N, S) \Leftrightarrow N > 2 \mid \text{sum}(N-2, S1'), S = 2*N-1+S1'.$$

Note that this rule r_1 cannot replace the original recursive rule because it only applies in case $N > 2$. It behaves like applying the original rule r twice. With rule r_1 we only need about half as many recursive steps as with the original rule alone. Because the arithmetic computation is simplified, we can also expect to halve the runtime.

We can now unfold rule r_1 with itself:

$$r_2 : \text{sum}(N, S) \Leftrightarrow N > 4 \mid \text{sum}(N-4, S1), S = 4 * N - 6 + S1$$

This rule results in fourfold speedup. We can continue this process, doubling the speed each time¹.

The most unfolded rule should cover as many recursive steps of the call as possible but not more. For example, for $N=4$ we will unfold till rule r_1 with guard $N>2$, for $N=5$ we will unfold till rule r_2 with $N>4$, for $N=50$ we will unfold till rule r_5 with $N>32$.

As we have just seen, our method requires unfolding on-the-fly because the number of unfoldings depends on the current call. We do not want to modify the given program at runtime. Therefore we

¹Clearly there is a closed form solution for this problem, $S = N * (N + 1)/2$, but this is not the point of the example.

also introduce a simple interpreter for the unfolded rules. This *meta-interpreter*² tries and applies each unfolded rule at most once starting with the given call and the most unfolded rule. With sufficient simplification of the unfolded rules (as in the example), a super-linear speedup in runtime can be achieved. The time complexity is reduced.

Overview and Contributions of the Paper. In this paper, we introduce our online program optimization strategy of runtime repeated recursion unfolding as a systematic way to enable significant speedups. We assume a single recursive rule with linear direct recursion and focus on tractable problems, i.e. those with polynomial worst-case time complexity. We will use summation as our running example.

Section 2 recalls syntax and semantics of the CHR programming language.

Section 3 defines our program transformation method of runtime repeated recursion unfolding with simplification and proves it correct. We also show that there is an optimal rule application strategy and prove it sound and complete.

Section 4 presents our lean implementation of the unfold and meta-interpreter to perform repeated recursion unfolding at runtime.

Section 5 derives the worst-case time complexity of our unfold and meta-interpreter in relation to that of the given recursive rule using recurrence equations.

Section 6 proves a sufficient condition as well as a sufficient and necessary condition for super-linear speedup relating the complexity of the recursive steps of the original rule and the unfolded rules.

Section 7 contains the experimental evaluation of our technique on three examples, summation, list reversal and sorting. We derive the necessary simplifications, analyse their time complexity and compare it with the result of benchmarks.

Section 8 discusses related work and *Section 9* discusses potential limitations and possible improvements of our approach. Finally, we end with conclusions and future work.

2. Preliminaries

We recall the abstract syntax and the equivalence-based abstract operational semantics of CHR (Constraint Handling Rules) [2] in this section.

2.1. Abstract Syntax of CHR

The CHR language is based on the abstract concept of constraints. *Constraints* are relations, distinguished predicates of first-order predicate logic. There are two kinds of constraints: *built-ins* (*built-in constraints*) and *user-defined* (*CHR*) *constraints* which are defined by the rules in a CHR program. Built-ins can be used as tests in the guard as well as for auxiliary computations in the body of a rule. There are at least the built-in constraints *true* and *false* (denoting inconsistency), including equality = over terms with arithmetic expressions and the usual relations over arithmetic expressions. When CHR is embedded into a host language, host language statements are regarded as built-ins.

²A meta-interpreter interprets a program written in its own implementation language.

Definition 2.1. (CHR Program and Rules)

A *CHR program* is a finite set of rules. A (*generalized simplification*) *rule* is of the form

$$r : H \Leftrightarrow C \mid B,$$

where r is an optional *name* (a unique identifier) of a rule. The *head* H is a conjunction of user-defined constraints, the optional *guard* C is a conjunction of built-ins, and the *body* B is a goal. The *local variables of a rule* are those not occurring in the head of the rule. A *goal* is a conjunction of built-in and user-defined constraints. A *call* is either an atomic constraint in a rule body or a given constraint. A *linear direct recursive rule* has exactly one call that has the same constraint symbol as the single head constraint.

(Possibly empty) conjunctions of constraints are denoted by upper case letters in definitions, lemmas and theorems. Conjunctions are understood as multisets of their atomic conjuncts. To avoid clutter, we often use simple commas to denote logical conjunction.

2.2. Abstract Operational Semantics of CHR

Computations in CHR are sequences of rule applications. The operational semantics of CHR is given by a state transition system where states are goals. It relies on an equivalence between states that abstracts from the representation of built-ins [3, 4]. Basically, two states are equivalent if their built-ins are logically equivalent (imply each other) and their user-defined constraints form equivalent multisets taking into account the built-ins. For example,

$$X \leq Y \wedge Y \leq X \wedge c(X, Y) \equiv X = Y \wedge c(X, X) \not\equiv X = Y \wedge c(X, X) \wedge c(X, X).$$

Let $\mathcal{CT}$ be a (decidable) constraint theory for the built-ins including equality $=$ over terms with arithmetic expressions. This means that arithmetic functions are interpreted and all other functions are not interpreted, ie. treated syntactically. A *copy* (*fresh variant*, *renaming*) of a rule is obtained by uniformly replacing its variables by new variables. We then say that the variables have been *renamed apart*.

Definition 2.2. (State Equivalence [3])

States are goals. Let C_i be the built-ins, let B_i denote user-defined constraints, and let $\mathcal{V}$ be a set of variables. Variables of a state that do not occur in $\mathcal{V}$ are called *local variables of the state*. Two states $S_1 = (C_1 \wedge B_1)$ and $S_2 = (C_2 \wedge B_2)$ with local variables $\bar{x}$ and $\bar{y}$, respectively, that have been renamed apart, are *equivalent*, written $S_1 \equiv_{\mathcal{V}} S_2$, if and only if

$$\mathcal{CT} \models \forall (C_1 \rightarrow \exists \bar{y} ((B_1 = B_2) \wedge C_2)) \wedge \forall (C_2 \rightarrow \exists \bar{x} ((B_1 = B_2) \wedge C_1))^3.$$

B_1 and B_2 are the multisets of user-defined constraints. They must be pairwise equivalent as enforced by $B_1 = B_2$. Note that in B_1 and B_2 we can freely replace a term t_1 by another term t_2 if the built-ins imply $t_1 = t_2$. Also, local variables occurring only in the built-ins can be removed (and introduced) if

³This definition implies $\mathcal{CT} \models \forall (\exists \bar{x} (B_1 \wedge C_1) \leftrightarrow \exists \bar{y} (B_2 \wedge C_2))$.

logical equivalence is maintained. Finally, all states with inconsistent built-ins are equivalent. These properties have been proven in [3]. An example illustrates some properties of state equivalence:

$$X=Y \wedge c(X, Y) \equiv_{\{X\}} c(X, X) \text{ but } X=Y \wedge c(X, Y) \not\equiv_{\{X, Y\}} c(X, X).$$

Using this state equivalence, the abstract CHR semantics is defined by a single transition (computation step) between states. It defines the application of a rule. If the source state can be made equivalent to a state that contains the head and the guard of a copy of a rule, then we can apply the rule by replacing the head by the body in the state. Any state that is equivalent to this target state is also in the transition relation.

Definition 2.3. (Transition and Computation)

A CHR transition (computation step) $S \mapsto_r T$ is defined as follows, where S is called *source state* and T is called *target state*:

$$\frac{S \equiv_{\mathcal{V}} (H \wedge C \wedge G) \not\equiv \text{false} \quad (r : H \Leftrightarrow C \mid B) \quad (C \wedge B \wedge G) \equiv_{\mathcal{V}} T}{S \mapsto_r T}$$

where the rule $(r : H \Leftrightarrow C \mid B)$ is a copy of a rule from a given program $\mathcal{P}$ such that its local variables do not occur in G . The goal G is called *context* of the rule application. It remains unchanged. It may be empty.

A *computation (derivation)* of a *query (given goal, call)* S with variables $\mathcal{V}$ in a program $\mathcal{P}$ is a connected sequence $S_i \mapsto_{r_i} S_{i+1}$ beginning with the query S as *initial state* S_0 and either ending in a *final state (answer, result)* S_n or otherwise *not terminating (diverging)*. The relation $\mapsto^*$ denotes the reflexive and transitive closure of $\mapsto$.

For convenience, we may drop the reference to the rules from the transitions. We may also drop $\mathcal{V}$ from the equivalence. Note that CHR is a committed-choice language, unlike Prolog there is no backtracking or undoing of rule applications.

Example 2.4. (Summation, Contd.)

Recall the rules for summation with *sum/2*:

$$\begin{aligned} b : \text{sum}(N, S) &\Leftrightarrow N = 1 \mid S = 1 \\ r : \text{sum}(N, S) &\Leftrightarrow N > 1 \mid \text{sum}(N-1, S1), S = N+S1 \end{aligned}$$

Then a computation for the query $\text{sum}(3, R)$ proceeds as follows.

$$\begin{aligned} \text{sum}(3, R) &\equiv_{\{R\}} \\ \text{sum}(N', S'), N' > 1, N'=3, S'=R &\mapsto_r \\ N' > 1, \text{sum}(N'-1, S1'), S' = N'+S1', N' = 3, S' = R &\equiv_{\{R\}} \\ \text{sum}(3-1, S1'), R = 3+S1' &\mapsto_r \\ \text{sum}(2-1, S1'), S1 = 2+S1', R = 3+S1 &\mapsto_b \\ S1'=1, S1 = 2+S1', R = 3+S1 &\equiv_{\{R\}} \\ R=6 \end{aligned}$$

3. Runtime Repeated Recursion Unfolding

We recall a definition of rule unfolding in CHR. Next we define simplification inside rule bodies. Then we have all the ingredients necessary to introduce runtime repeated recursion unfolding and show its correctness. We also prove some useful lemmas. We also show that there is a straightforward optimal rule application strategy and prove it sound and complete.

We will need the standard notions of substitutions, matching and instances. A *substitution* is a mapping function from variables to terms $\theta : \mathcal{V} \rightarrow \mathcal{T}$, written in postfix notation, such that domain of θ , the set $\text{dom}(\theta) = \{X \mid X\theta \neq X\}$, is finite. When a substitution is applied to a goal, it is applied to all variables in the goal. If $A = B\theta$, where B is a goal, we say that A is an *instance* of B , A *matches* B , and that B is *instantiated*.

3.1. Rule Unfolding

For unfolding of rules in CHR, we follow the definition and proofs of [5]. In this paper we rewrite their definition of unfolding in terms of generalized simplification rules. This simplifies the definition and is sufficient for our purposes.

To define unfolding, we need the following notation. For a goal A , let $\text{vars}(A)$ denote the set of variables in A . Set difference $C_1 = C_2 \setminus C_3$ for conjunctions of built-ins is defined as $C_1 = \{c \in C_2 \mid C\mathcal{T} \not\models C_3 \rightarrow c\}$. In words, to obtain C_1 , remove from C_2 the built-in constraints that C_3 implies.

Definition 3.1. (Unfolding (based on Def. 8 [5]))

Let $\mathcal{P}$ be a CHR program and let $r, v \in \mathcal{P}$ be two rules whose variables have been renamed apart

$$\begin{aligned} r : H &\Leftrightarrow C \mid D \wedge B \wedge G \\ v : H' &\Leftrightarrow C' \mid B', \end{aligned}$$

where D is the conjunction of the built-ins in the body of r . Then we define the *unfolding of rule r with rule v*

$$\text{unfold}(r, v) = r'$$

as follows. Let θ be a substitution such that $\text{dom}(\theta) \subseteq \text{vars}(H')$. Let $C''\theta = C'\theta \setminus (C \wedge D)$. If $C\mathcal{T} \models \exists(C \wedge D) \wedge \forall((C \wedge D) \rightarrow G=H'\theta)$, $\text{vars}(C''\theta) \cap \text{vars}(H'\theta) \subseteq \text{vars}(H)$ and $C\mathcal{T} \models \exists(C \wedge C''\theta)$, then the unfolded rule r' is

$$r' : H \Leftrightarrow C \wedge C''\theta \mid D \wedge B \wedge G=H' \wedge B'.$$

If a goal G in the body of rule r matches the head H' of a rule v , unfolding replaces G by the body of rule v together with $G=H'$ to obtain a new rule r' . We also add to its guard C an instance of a part of the guard of rule v . This part C'' contains the non-redundant built-ins of C' (they are not implied by the built-ins in the rule r).

Note that for a correct unfolding according to the above definition, three conditions have to be met. The chosen substitution must make H' equivalent to the matching G in the context of the built-ins of rule r that must be satisfiable. Under this substitution, the common variables of H' and C'' must

already occur in H , and finally the guard of the unfolded rule must be satisfiable. If these conditions are violated, unfolding cannot take place and no unfolded rule is produced.

Correctness of unfolding means that the unfolded rule does not lead to new states when it is applied, it is redundant.

Lemma 3.2. (Correctness of Unfolding)

Given a CHR program with rules r and v and their unfolding resulting in rule $r' = \text{unfold}(r, v)$ and a computation with a transition that applies the unfolded rule $G \mapsto_{r'} G'$. Then there exists a computation where we replace the transition by a sequence of transitions without the unfolded rule $G \mapsto^* G'$ and leave all other states and transitions unchanged.

Proof. Correctness of unfolding is proven in *Corollary 1* [5]. □

In that sense, a correctly unfolded rule is always redundant (but of course its application is expected to improve efficiency).

Lemma 3.3. (Redundancy of Unfolded Rules)

Given the rules r and v and their unfolding resulting in rule $r' = \text{unfold}(r, v)$ and any goal G with a transition with the unfolded rule

$$G \mapsto_{r'} G'',$$

then there exist transitions with the original rules either of the form

$$G \mapsto_r G' \mapsto_v G'' \text{ or } G \mapsto_r G'' \equiv \text{false}.$$

Proof. The lemma corresponds to *Proposition 6* in the appendix of [5], where the proof can be found. □

Example 3.4. (Summation, contd.)

We unfold the recursive rule for summation with (a copy of) itself:

$$\begin{aligned} r : \text{sum}(N, S) &\Leftrightarrow N > 1 \mid S = N + S1, \text{sum}(N-1, S1) \\ v : \text{sum}(N', S') &\Leftrightarrow N' > 1 \mid S' = N' + S1', \text{sum}(N'-1, S1') \end{aligned}$$

Then the unfolded rule is

$$\begin{aligned} r_1 : \text{sum}(N, S) &\Leftrightarrow N > 1, N-1 > 1 \mid S = N + S1, \text{sum}(N-1, S1) = \text{sum}(N', S'), \\ &S' = N' + S1', \text{sum}(N'-1, S1') \end{aligned}$$

Unfolding is possible since its three conditions are met. First, $\text{sum}(N-1, S1)$ is an instance of $\text{sum}(N', S')$, more precisely

$$(N > 1, S = N + S1) \rightarrow \text{sum}(N-1, S1) = \text{sum}(N', S')\theta,$$

where the substitution θ maps N' to $N-1$ and S' to $S1$. Second,

$$\text{vars}(N-1 > 1) \cap \text{vars}(\text{sum}(N-1, S1)) \subseteq \text{vars}(\text{sum}(N, S))$$

holds since $\{N\} \cap \{N, S1\} \subseteq \{N, S\}$. Third, the new guard $N > 1, N-1 > 1$ is satisfiable.

Obviously we can simplify the built-ins of the guard and the body of this rule, and we will define this kind of simplification next.

3.2. Rule Simplification

Speedup crucially depends on the amount of simplification that is possible in the unfolded rules. We want to replace built-ins by semantically equivalent ones that can be executed more efficiently. We define a suitable notion of rule simplification and prove it correct. In this subsection, we basically follow [1].

Definition 3.5. (Rule Simplification)

Given a rule r of the form

$$r : H \Leftrightarrow C \mid D \wedge B,$$

where D are the built-ins and B are the user-defined constraints in the body of the rule. We define

$$\begin{aligned} \text{simplify}(r) &= (H' \Leftrightarrow C' \mid D' \setminus C' \wedge B') \text{ such that} \\ (H \wedge C) &\equiv_{\mathcal{V}} (H' \wedge C') \text{ and } (C \wedge D \wedge B) \equiv_{\mathcal{V}} (D' \wedge B'), \end{aligned}$$

where C' and D' are the built-ins and H' and B' are the user-defined constraints, where $\mathcal{V} = \text{vars}(H) \cup \text{vars}(H')$, and $\text{simplify}(r)$ is simpler than r according to some strict partial order.

In the given rule, we replace head and guard, and the body, respectively, by simpler yet state equivalent goals. We may remove redundant constraints, we may replace constraints by more efficient ones. The choice of $\mathcal{V}$ allows us to remove local variables if possible, i.e those that occur only in the guard or body of the rule. We temporarily add the guard C when we simplify the body for correctness and to improve the simplification. What is simpler depends on the particular built-in constraints used.

For correctness we have to show that the same transitions $S \mapsto T$ are possible with rule r and rule $\text{simplify}(r)$.

Theorem 3.6. (Correctness of Rule Simplification)

(Theorem 1 of [1]) Let $r = (H \Leftrightarrow C \mid D \wedge B)$ be a rule and let $s = (H' \Leftrightarrow C' \mid D' \setminus C' \wedge B')$ be the simplified rule $s = \text{simplify}(r)$. For any state S and variables $\mathcal{V}$, $S \mapsto_r T$ iff $S \mapsto_s T$.

Proof. According to the definition of a CHR transition (Def. 2.3) and of rule simplification (Def. 3.5), we know that

$$\begin{aligned} S \mapsto_r T &\text{ iff } S \equiv_{\mathcal{V}} (H \wedge C \wedge G) \not\equiv \text{false} \text{ and } (C \wedge D \wedge B \wedge G) \equiv_{\mathcal{V}} T \\ S \mapsto_s T &\text{ iff } S \equiv_{\mathcal{V}} (H' \wedge C' \wedge G') \not\equiv \text{false} \text{ and } (C' \wedge D' \setminus C' \wedge B' \wedge G') \equiv_{\mathcal{V}} T \\ &\quad (H \wedge C) \equiv_{\mathcal{V}'} (H' \wedge C') \\ &\quad (C \wedge D \wedge B) \equiv_{\mathcal{V}'} (D' \wedge B'), \end{aligned}$$

where $\mathcal{V}' = \text{vars}(H) \cup \text{vars}(H')$.

It suffices to show that $S \mapsto_r T$ implies $S \mapsto_s T$, since the implication in the other direction is symmetric and can be shown in the same way.

Hence we have to show that there exists a goal G' such that

$$\begin{aligned} S &\equiv (H \wedge C \wedge G) \equiv_{\mathcal{V}} (H' \wedge C' \wedge G') \text{ if } (H \wedge C) \equiv_{\mathcal{V}'} (H' \wedge C') \text{ and} \\ T &\equiv (C \wedge D \wedge B \wedge G) \equiv_{\mathcal{V}} (C' \wedge D' \setminus C' \wedge B' \wedge G') \text{ if } (C \wedge D \wedge B) \equiv_{\mathcal{V}'} (D' \wedge B'). \end{aligned}$$

We choose $G' = C \wedge G$. Note that $(C' \wedge D' \setminus C')$ is just $(C' \wedge D')$. The main part of the proof reasons on the first-order logic formulas resulting from applying the definition of state equivalence (Def. 2.2) to the above equivalences. The full proof can be found in appendix A of the full version of [1]. $\square$

We conclude this subsection by simplification of the unfolded rule of our running example.

Example 3.7. (Summation, contd.)

Recall the unfolded rule

$$\begin{aligned} \text{sum}(N, S) \Leftrightarrow N > 1, N-1 > 1 \mid S = N + S1, \text{sum}(N-1, S1) = \text{sum}(N', S'), \\ S' = N' + S1', \text{sum}(N'-1, S1'). \end{aligned}$$

For the head and guard we have that

$$\text{sum}(N, S), N > 1, N-1 > 1 \equiv_{\{S, N\}} \text{sum}(N, S), N > 2.$$

For the body we have that

$$\begin{aligned} N > 1, N-1 > 1, S = N + S1, \text{sum}(N-1, S1) = \text{sum}(N', S'), S' = N' + S1', \text{sum}(N'-1, S1') \equiv_{\{S, N\}} \\ N > 2, S = 2 * N - 1 + S1', \text{sum}(N-2, S1'). \end{aligned}$$

Thus the unfolded rule can be simplified into the rule

$$\text{sum}(N, S) \Leftrightarrow N > 2 \mid S = 2 * N - 1 + S1', \text{sum}(N-2, S1').$$

3.3. Runtime Repeated Recursion Unfolding

We can now define our novel program optimization strategy of runtime repeated recursion unfolding based on rule unfolding and rule simplification. We prove it correct by showing the redundancy of unfolded recursive rules and their termination. On the way, we will also prove lemmas about the number of recursive steps covered and the number of rules generated.

In our method, we start from a call (query) for a CHR constraint defined by a recursive rule. We unfold the recursive rule with itself and simplify it. Then we unfold the resulting rule. We repeat this process as long as the resulting rules are applicable to the query. In this paper, we assume a single linear direct recursive rule.

Definition 3.8. (Runtime Repeated Recursion Unfolding)

Let r be a recursive rule and G be a goal. Let

$$\text{unfold}(r) = \text{unfold}(r, r).$$

The *runtime repeated recursion unfolding* of a recursive rule r with goal G and with rule simplification is a maximal sequence of rules $r_0, r_1, \dots$ where

$$\begin{aligned} r_0 &= r \\ r_{i+1} &= \text{simplify}(\text{unfold}(r_i)) \text{ if } G \mapsto_{r_{i+1}} G', (i \geq 0) \end{aligned}$$

The definition describes the repetition of the following step to produce the desired sequence of more and more unfolded rules: We unfold and simplify the current unfolded rule r_i . If the unfolding is possible and if the resulting rule r_{i+1} is applicable to the query G (as expressed by $G \mapsto_{r_{i+1}} G'$), we add the new rule to the sequence and continue with it.

Example 3.9. (Summation, contd.)

Consider a query $sum(10, R)$. Recall the unfolded simplified rule

$$r_1 = sum(N, S) \Leftrightarrow N > 2 \mid S = 2 * N - 1 + S1, sum(N-2, S1).$$

Since $sum(10, R) \mapsto_{r_1} 10 = N, N > 2, \dots$, we repeat the unfolding:

$$\begin{aligned} unfold(r_1) &= sum(N, S) \Leftrightarrow N > 2, N-2 > 2 \mid S = 2 * N - 1 + S1, \\ sum(N-2, S1) &= sum(N', S'), S' = 2 * N' - 1 + S1', sum(N'-2, S1'). \end{aligned}$$

The unfolded rule can be simplified into the rule

$$simplify(unfold(r_1)) = r_2 = sum(N, S) \Leftrightarrow N > 4 \mid S = 4 * N - 6 + S1', sum(N-4, S1').$$

The rule r_2 is applicable to the goal. Further recursion unfolding results in rules with guards $N > 8$ and then $N > 16$. To the latter rule, the goal $sum(10, R)$ is not applicable anymore. Hence runtime repeated recursion unfolding stops. The rules for the goal $sum(10, R)$ are therefore (more unfolded rules come first):

$$\begin{aligned} r_3 &= sum(N, S) \Leftrightarrow N > 8 \mid S = 8 * N - 28 + S1, sum(N-8, S1) \\ r_2 &= sum(N, S) \Leftrightarrow N > 4 \mid S = 4 * N - 6 + S1, sum(N-4, S1) \\ r_1 &= sum(N, S) \Leftrightarrow N > 2 \mid S = 2 * N - 1 + S1, sum(N-2, S1) \\ r &= r_0 = sum(N, S) \Leftrightarrow N > 1 \mid S = N + S1, sum(N-1, S1) \\ b &= sum(N, S) \Leftrightarrow N = 1 \mid S = 1. \end{aligned}$$

Note that to the goal $sum(10, R)$ we can apply any of the recursive rules. The most efficient way is to start with the first, most unfolded rule. It covers more recursive steps of the original recursive rule than any other rule. We will formalize such optimal rule applications in the next section.

We now prove some useful properties of runtime repeated recursion unfolding. Unfolded recursive rules are redundant. As is the case for any unfolded rule, their computations can also be performed with the original rule.

Lemma 3.10. (Redundancy of Unfolded Recursive Rules)

Assume a runtime repeated recursion unfolding of a recursive rule r with goal G . It results in a sequence of rules $r_0, \dots, r_i, \dots$ where $r = r_0, i \geq 0$.

Then for any goal B with a transition

$$B \mapsto_{r_{i+1}} B''$$

there exist transitions either of the form

$$B \mapsto_{r_i} B' \mapsto_{r_i} B'' \text{ or } B \mapsto_{r_i} B'' \equiv \text{false}.$$

Proof. This claim follows immediately from (Lemma 3.3). $\square$

One computation step (transition) with an unfolded rule corresponds to two computation steps with the rule that was unfolded (if no inconsistency is involved). So each unfolded rule doubles the number of recursive steps of the original rule that it covers.

Lemma 3.11. (Recursive Steps Covered by Unfolded Recursive Rules)

Assume runtime repeated recursion unfolding of a recursive rule r with goal G . It results in a sequence of rules $r_0, \dots, r_i, \dots$ where $r = r_0, i \geq 0$. If

$$G \mapsto_{r_i} G' \text{ with } G' \not\equiv \text{false},$$

then there exists a sequence of 2^i transitions with rule r

$$G \mapsto_r G_1 \dots \mapsto_r G_{2^i} \text{ with } G' \equiv G_{2^i}.$$

Proof. By correctness of rule simplification (Theorem 3.6), rule r and its simplification $\text{simplify}(r)$ admit equivalent transitions. We can therefore ignore the application of rule simplification (cf. Definition 3.8) in this proof.

We will use induction over the rule index j ($i \geq j \geq 0$), going from the largest unfolded rule r_i to the original rule r_0 . We actually prove a more general result: that with rule r_j we need 2^{i-j} transitions. We first consider the base case. Our claim holds trivially for $j = i$ resulting in 2^0 , i.e. one transition with rule r_i .

For the induction argument, we assume for rule r_{j+1} we need $2^{i-(j+1)}$ transitions. Then for rule r_j we claim to need twice as many, 2^{i-j} transitions. This can be shown by replacing each transition $B \mapsto_{r_{j+1}} B''$ by the two transitions $B \mapsto_{r_j} B' \mapsto_{r_j} B''$ according to (Lemma 3.10).

The lemma also admits another possible replacement $B \mapsto_{r_i} B'' \equiv \text{false}$. But all states in any computation starting with G and ending in $G' \equiv G_{2^i}$ are different from false because $G' \not\equiv \text{false}$ and no transition is possible from a state false . So the replacement involving false is not possible.

Thus for $j = 0$, i.e. rule $r_0 = r$, we need 2^i transitions for one transition with rule r_i . $\square$

Hence rule r_i covers 2^i recursive steps of the original recursive rule r with goal G if the computation does not end in a state false .

For the upcoming lemmas, we define when a goal G takes n recursive steps with the original recursion.

Definition 3.12. (Recursion Depth of a Goal)

Given a goal G with a recursive rule r . Let n be the maximum number of transitions starting from the query G that only involve applications of the given recursive rule r

$$G \mapsto_r G_1 \dots \mapsto_r G_n \text{ and there is no transition with } r \text{ from } G_n.$$

If the computation is finite and terminates, then we call n the recursion depth of goal G with rule r .

We can unfold rules as long as the number of recursive steps they cover does not exceed n . This gives us a limit on the number of rules that we can generate.

Lemma 3.13. (Number of Unfolded Recursive Rules)

Given a goal G with a recursive rule r that has recursion depth n and ends in a state $G_n \neq \text{false}$. Then repeated recursion unfolding will generate k rules such that $2^k \leq n$. Hence, $k \leq \lfloor \log_2(n) \rfloor$.

Proof. By contradiction: Assume repeated recursion unfolding generates a rule r_k such that $2^k > n$. According to *Lemma 3.11* rule r_k allows for a transition with G that is equivalent to 2^k transitions with the original recursive rule r . But the maximum number of transitions possible with r is just n . $\square$

Note that fewer rules than $\lfloor \log_2(n) \rfloor$ may be generated because (further) unfolding is not possible if its three conditions are not met.

Lemma 3.14. (Termination of Runtime Repeated Recursion Unfolding)

Given a goal G with a recursive rule r that has recursion depth n and ends in a state $G_n \neq \text{false}$. Then the runtime repeated recursion unfolding of r with G terminates.

Proof. Direct consequence of *Lemma 3.13*. $\square$

So we can ensure that runtime repeated recursion unfolding terminates with a goal if the original recursive rule terminates with that goal.

We give two simple examples for nontermination.

Example 3.15. (Nontermination)

The goal $p(0)$ does not terminate with the recursive rule:

$$r : p(N) \Leftrightarrow N \neq 1 \mid p(N-1).$$

Runtime repeated recursion unfolding with goal $p(0)$ results in the rule

$$r_1 : p(N) \Leftrightarrow N \neq 1, N \neq 2 \mid p(N-2).$$

Since the rule is applicable to the goal $p(0)$, our unfolding can proceed. Each unfolding adds an inequality to the guard, but the guards will always admit $N=0$. Therefore, runtime repeated recursion unfolding does not terminate as well.

The next example shows that the condition that the resulting state is not *false* is necessary. We use a variation of the rule above.

Example 3.16. (Nontermination with *false*)

The goal $p(0)$ terminates in a state *false* when applying the following recursive rule,

$$r : p(N) \Leftrightarrow N \neq 1 \mid N < 0, p(N-1),$$

since the body built-in $N < 0$ is inconsistent with $N=0$ from the goal $p(0)$. The unfolded and simplified rule is

$$r_1 : p(N) \Leftrightarrow N \neq 1, N \neq 2 \mid N < 0, p(N-2).$$

Again, with the goal $p(0)$, unfolding can proceed forever. Runtime repeated recursion unfolding does not terminate even though the computation with the original rule r terminated. Still, for the goal $p(0)$ any computation with any unfolded rule will lead to *false*.

Based on the lemmas proven, we can now directly show correctness of our method.

Theorem 3.17. (Correctness of Runtime Repeated Recursion Unfolding)

Given a goal G with a recursive rule r that has recursion depth n and ends in a state $G_n \neq \text{false}$. Then the runtime repeated recursion unfolding of rule r with goal G terminates and generates redundant unfolded rules.

Proof. The claim is a direct consequence of termination proven in *Lemma 3.14* and the redundancy of unfolded recursive rules proven in *Lemma 3.10*. $\square$

3.4. Optimal Rule Applications

An unfolded rule covers twice as many recursion steps than the given rule. When we apply a more unfolded rule, we cover more recursive steps with a single rule application. Based on this observation we introduce a rule application strategy where we try to apply more unfolded rules first. Furthermore each unfolded rule is tried only once and is applied at most once. We prove our optimal rule application strategy sound and complete.

Definition 3.18. (Optimal Rule Application Strategy)

Given a recursive rule r_0 with a goal G with k additional rules $r_0, r_1, \dots, r_{k-1}, r_k$ from runtime repeated recursion unfolding. Let the notation $G_i \mapsto_r^{opt} G'$ be shorthand for $G_i \mapsto_r G'$ if $G' \neq \text{false}$ or otherwise $G_i \equiv G'$. Then the *optimal rule application strategy* is as follows:

$$G \mapsto_{r_k}^{opt} G_k \mapsto_{r_{k-1}}^{opt} G_{k-1} \dots G_2 \mapsto_{r_1}^{opt} G_1 \mapsto_{r_0}^{opt} G_0.$$

As a result of this strategy, to the query G we apply the most unfolded rule r_k exactly once⁴. In the remaining computation, no matter if a rule r_i ($i < k$) was applied or not, we next try to apply rule r_{i-1} until $i=0$.

We first show soundness of this computation strategy. Computations with optimal rule applications correspond to computations with the original rule only.

Theorem 3.19. (Soundness of Optimal Rule Applications)

Given a recursive rule r_0 with a goal G with k additional rules from runtime repeated recursion unfolding.

Then for a computation for goal G with optimal rule applications there exists a computation for G only using the original recursive rule r that ends in an equivalent state.

Proof. In such a computation, by *Lemma 3.10*, we can replace a transition with rule r_{i+1} ($0 \leq i < k$) by transitions with only rule r_i . Furthermore the resulting states of these computations are equivalent. Thus we can repeat this process of replacement until all transitions only involve rule r and the computation will end in an equivalent state. $\square$

⁴We know the application is possible since otherwise the unfolding would not have taken place.

We have seen that a transition with an unfolded rule can replace transitions with the original rule. The other direction is not necessarily true. The unfolded rule may not be applicable because the guard of the unfolded rule may come out stricter than necessary. For our optimal rule application strategy to be complete, we require that unfolding generates all rules with the following property: If a rule can perform two recursive transitions for a goal, then its unfolded rule is also applicable to the goal.

Theorem 3.20. (Completeness of Optimal Rule Applications)

Let r_0 be a recursive rule with a goal G with recursion depth n with $k = \lfloor \log_2(n) \rfloor$ rules from runtime repeated recursion unfolding, where for any rule r_i ($0 \leq i < k$) and any goal B with transitions $B \mapsto_{r_i} B' \mapsto_{r_i} B''$ there exists a transition $B \mapsto_{r_{i+1}} B''$.

Then for any computation for G with rule r_0 with recursion depth n there exists a computation for G with optimal rule applications that ends in an equivalent state.

Proof. We start from a computation only using rule r_0 . According to the condition in the claim, we can replace the first two transitions with rule r_0 by one transition with rule r_1 without changing the resulting state. We repeat this for the remaining pairs of subsequent transitions. We get a computation with transitions using rule r_1 ending in at most one transition with rule r_0 . With rule r_1 we start from the first transition again and repeat this process of replacing two transitions by one of rule r_2 . We continue going from rule r_i to rule r_{i+1} until $i + 1 = k$. But now we have a computation that applies each rule from $r_k, r_{k-1}, \dots, r_1, r_0$ at most once and in the given order of the rules. So this computation is one with optimal rule applications. $\square$

Example 3.21. (Summation, contd.)

Recall that the rules for $sum/2$ are:

$$\begin{aligned} r_3 &= sum(N, S) \Leftrightarrow N > 8 \mid S = 8 * N - 28 + S1, sum(N-8, S1) \\ r_2 &= sum(N, S) \Leftrightarrow N > 4 \mid S = 4 * N - 6 + S1, sum(N-4, S1) \\ r_1 &= sum(N, S) \Leftrightarrow N > 2 \mid S = 2 * N - 1 + S1, sum(N-2, S1) \\ r &= r_0 = sum(N, S) \Leftrightarrow N > 1 \mid S = N + S1, sum(N-1, S1) \\ b &= sum(N, S) \Leftrightarrow N = 1 \mid S = 1. \end{aligned}$$

A computation with optimal rule applications for the goal $sum(10, R)$ is:

$$\begin{aligned} sum(10, R) &\mapsto_{r_3} \\ 10 = N, N > 8, R = S, S &= 8 * N - 28 + S1, sum(N-8, S1) \equiv_{\{R\}} \\ R &= 52 + S1, sum(2, S1) \mapsto_{r_0} \\ R &= 52 + S1, 2 = N', N' > 1, S1 = S', S' = N' + S1', sum(N'-1, S1') \equiv_{\{R\}} \\ R &= 54 + S1', sum(1, S1') \mapsto_b \\ R &= 55 \end{aligned}$$

4. Implementation of Runtime Repeated Recursion Unfolding

We introduce the implementation of our runtime program transformation. At compile-time, the rules for the given *recursive constraint* are replaced by a call to the *unfolder* that contains these rules and

then to the meta-interpreter that interprets the unfolded rules. At runtime, the unfolders repeatedly unfolds a recursive rule as long as it is applicable to a given goal using a predefined *unfolding scheme* that includes the simplification. Then the meta-interpreter applies the resulting unfolded rules according to the optimal rule application strategy. We use an interpreter because we do not want to modify the given program at runtime.

For our implementation, we use CHR embedded in Prolog. Such sequential CHR systems execute the constraints in a goal from left to right and apply rules top-down according to their textual order in the program. A user-defined constraint in a goal can be understood as a procedure call that traverses the rules of the program. If it and possibly previous constraints from the goal match the head of a rule, a copy of the rule is instantiated according to the matching. If the guard check of the rule copy holds, then the rule is applicable. For application, the matched constraints are replaced by the body of the rule copy and execution continues with the calls in the body. The first applicable rule will be applied, and this application cannot be undone, it is committed-choice (in contrast to clause application in Prolog). This behavior has been formalized in the so-called *refined semantics* which is a proven concretization of the abstract operational semantics [6].

According to the CHR semantics, all Prolog predicates are regarded as built-in constraints. In the following code in concrete syntax, `=/2`, `copy_term/2` and `call/1` are standard built-in predicates of Prolog. The syntactic equality `=/2` tries to unify its arguments, i.e. making them syntactically identical by instantiating their variables appropriately. The built-in `copy_term/2` produces a copy (variant, renaming) of the given term with new fresh variables. The arithmetic equality `is/2` tries to unify its first argument with the result of evaluating the arithmetic expression in its second argument. The Prolog meta-call `call/1` executes its argument as a goal. It works for both Prolog built-in predicates and CHR constraints. Our implementation with CHR in SWI Prolog [7, 8] together with the examples and benchmarking code is available online at <https://sp2.informatik.uni-ulm.de/fruehwirth/rrru.pl>.

4.1. Unfolder Implementation

The unfolders is implemented as a recursive CHR constraint `unf/3`. It repeatedly unfolds and simplifies a recursive rule as long as it is applicable to a goal. In `unf(G,Rs,URs)`, the first argument `G` is the goal and `R`s is a list of rules. `UR`s is the resulting list of unfolded rules. We assume that in the goal `G` the input arguments are given and the output arguments are variables. Initially, the list `R`s consists of the recursive rule followed by one or more rules for the base cases of the recursion. Consider the code below. The comment in the first line declares the arguments of `unf/3` as either input (+) or output (-). A variable that occurs only once in a CHR rule has a name that starts with an underscore character.

```
% unf(+RecursiveGoal, +RuleList, -UnfoldedRuleList)

unf(G, [R|Rs], URs) <=>                % recursive step
  R = (H <=> Co | _),                    % get head H and guard Co of rule R
  copy_term((H <=> Co),(G <=> C)),        % copy them, unify head copy with goal G
  call(C)                               % call instantiated guard copy C
  |
  simp_unf(R, UR),                      % unfold and simplify rule R into rule UR
```

```

unf(G, [UR,R|Rs], URs).           % add new rule UR and recurse

unf(_G, [_R|Rs], URs) <=> URs=Rs. % otherwise return rules Rs in URs

```

In a recursive step of `unf/3`, the first rule element in the list `Rs` is unfolded and added in front of `Rs`. In the base case of the recursion, the final resulting list of unfolded and given rules is returned in `URs`. We rely on the refined CHR semantics and its rule order to ensure that the rule for the base case is only applied if the recursive rule is not applicable.

We explain the recursive rule for `unf/3` in detail now. We check if the rule `R` in the list is applicable to the query (call, goal) `G`. The guard check is performed by getting (using `=/2`) and copying the relevant parts (head and guard) of rule `R`, unifying the copied head with the goal (all with `copy_term/2`) and then executing the instantiated guard copy with `call/1`. The copies will not be needed after that.

If the guard check succeeds, we unfold the current rule `R` with itself and simplify it using `simp_unf/2` and add the resulting rule `UR` to the rule list in the recursive call of `unf/3`. Note that we unfold the given general rule, not the instance of the rule stemming from the query.

The Prolog predicate `simp_unf/2` implements the *unfolding scheme*. Its call `simp_unf(R,UR)` takes the current rule $R=r_i$ and computes its simplified unfolding $UR=r_{i+1}$ according to $r_{i+1} = \text{simplify}(\text{unfold}(r_i))$ in Definition 3.8. For ease of implementing `simp_unf/2`, we use a *rule template* t_r which is a suitable generalization of the given recursive rule $r=r_0$ and its simplified unfoldings r_i . The rules are then instances of the template, i.e. $r_i = t_r\theta_i$. The substituted variables $\text{dom}(\theta_i)$ in the template represent the *parameters* for the instance. The parameters will be bound at runtime. Therefore the head of the clause for `simp_unf/2` will be of the form `simp_unf( $t_r, t'_r$ )`. In the body of `simp_unf/2`, the parameters for the unfolded rule will be computed from the parameters of the current rule where r_i and r_{i+1} will be an instance of t_r and t'_r .

When the guard check has failed, the base case of `unf/3` returns the rules that have been accumulated in the rule list as the result list in the third argument (with the exception of the first rule to which the goal was not applicable).

To simplify the implementation, the body of the rules in the lists syntactically always consists of three conjuncts of goals: the constraints before the recursive goal, the recursive goal and the constraints after the recursive goal. If there are no such constraints (or no recursive goal in the base case), we use the built-in `true` to denote an empty conjunct.

The following example clarifies the above remarks on the implementation.

Example 4.1. (Summation, contd.)

We show how we implement `unfold` and `simplify` with `simp_unf/2` for the summation example. We abbreviate `sum` to its first letter `s` to avoid clutter in the code. The rule template for `sum` is

```

s(A,C) <=> A>V | B is A-V, s(B,D), C is V*A-W+D   % rule template

```

where the variables `V` and `W` are parameters that stand for integers. Its instance for the original recursive rule is

```

s(A,C) <=> A>1 | B is A-1, s(B,D), C is 1*A-0+D   % rule instance V=1, W=0

```

The implementation of the unfolding scheme for summation is accomplished by the following Prolog clause for `simp_unf/2`.

```
simp_unf(
  (s(A,C) <=> A>V | B is A-V, s(B,D), C is V*A-W+D),
  (s(A1,C1) <=> A1>V1 | B1 is A1-V1, s(B1,D1), C1 is V1*A1-W1+D1)
) :-
  V1 is 2*V, W1 is 2*W+V*V.
```

For a goal `s(100,S)` the unfolders are called with

```
unf(s(100,S), [
  (s(A,C)<=>A>1| B is A-1, s(B,D), C is 1*A-0+D), % original recursion
  (s(A,B)<=>A=1| B=1, true, true)                  % base case
], URs).
```

It will return the following rules in the list `URs`:

```
s(A,C) <=> A>64 | B is A-64, s(B, D), C is 64*A-2016+D
s(A,C) <=> A>32 | B is A-32, s(B, D), C is 32*A-496+D
s(A,C) <=> A>16 | B is A-16, s(B, D), C is 16*A-120+D
s(A,C) <=> A>8 | B is A-8, s(B, D), C is 8*A-28+D
s(A,C) <=> A>4 | B is A-4, s(B, D), C is 4*A-6+D
s(A,C) <=> A>2 | B is A-2, s(B, D), C is 2*A-1+D
s(A,C) <=> A>1 | B is A-1, s(B, D), C is 1*A-0+D      % original recursion
s(A,C) <=> A=1 | C=1, true, true                       % base case
```

4.2. Meta-Interpreter Implementation

We implement the optimal rule application strategy with the help of a meta-interpreter for CHR. Our meta-interpreter handles the recursive calls, any other goal will be handled by the underlying CHR implementation. To a recursive goal, the meta-interpreter tries to apply the unfolded rules produced by the unfolders and applies them at most once. The meta-interpreter is called with `mip(G,Rs)`, where `G` is the given recursive goal and `R`s is the list of rules from the unfolders `unf/3`.

```
% mip(+RecursiveGoal, +RuleList)

mip(true,_Rs) <=> true. % base case, no more recursive goal
mip(G,[R|Rs]) <=>      % current rule is applicable to goal G
  copy_term(R, (G <=> C | B,G1,D)), % copy rule, unify head copy with G
  call(C)                          % check guard
  |
  call(B),                          % execute constraints before recursive call
  mip(G1,Rs),                       % recurse with recursive goal and remaining rules
  call(D).                          % execute constraints after recursive call
```

```

mip(G,[_R|Rs]) <=>      % current rule is not applicable
  mip(G,Rs).             % try remaining rules on G

```

We now discuss the three rules of our meta-interpreter.

- In the first rule, the base case is reached since the recursive goal has been reduced to true.
- The second meta-interpreter rule tries to apply the rule R in the rule list to the current goal G. It copies the rule, unifies the copied head with the goal and then checks if the guard C holds with a meta-call. If so, the rule is applied. The conjunct before the recursive goal B is directly executed with a meta-call. Next, the recursive goal G1 is handled with a recursive call to the meta-interpreter using the remainder of the rule list. Finally the conjunct after the recursive goal D is directly executed with a meta-call.
- Otherwise the first rule from the rule list was not applicable (according to the refined semantics), and then the last meta-interpreter rule recursively continues with the remaining rules in the list.

This ensures that each unfolded rule is tried and applied at most once in accordance with the optimal rule application strategy.

4.3. Recursive Constraint Implementation

In order to enable runtime repeated recursion unfolding, at compile-time, the rules for the given recursive constraint c/k are replaced by a call to the unfold `unf/3` that contains these rules and then to the meta-interpreter `mip/2` that interprets the unfolded rules. We replace according to the rule template named `rec_unfold` where $X_1, \dots, X_k$ are different variables and `OriginalRules` is the list of the given original rules that defined the recursive constraint.

```

% rule template for a recursive constraint c/k

rec_unfold @ c(X1,...,Xk) <=>
  unf(c(X1,...,Xk), OriginalRules, UnfoldedRules),
  mip(c(X1,...,Xk), UnfoldedRules).

```

Example 4.2. (Summation, contd.)

For the summation example, the `rec_unfold` rule instance is as follows:

```

rec_unfold @ sum(N,S) <=>
  unf(s(N,S), [
    (s(A,C) <=> A>1 | B is A-1, s(B, D), C is 1*A-0+D),
    (s(A,C) <=> A=1 | C=1, true, true)
  ], URs),
  mip(s(N,S), URs).

```

5. Time Complexity of the Implementation

For the worst-case time complexity of our implementation of runtime repeated recursion unfolding, we have to consider the recursion in the original rule, and the recursions in the unfolders as well as meta-interpreter. We parametrize the time complexity by the number of recursive steps with the original rule. From the time complexity of the recursive step we can derive the time complexity of the recursion using recurrence equations. This gives us a precise measure of the complexity of the recursion. We assume some familiarity with the T-notation for time complexity (cf. Chapter 2 in [9]) as well as stating and solving recurrences (cf. Chapter 4 in [9]).

Our time complexity considerations are based on [10] and on the following realizable assumptions for the Prolog built-in predicates: Matching, unification and copying take constant time for given terms and linear time in the size of the involved terms in general. A Prolog meta-call has the same time complexity as directly executing its goal argument.

In the following annotated code for the unfolders and meta-interpreter, the comments indicate the time complexity of each non-recursive goal in the bodies of the rules. A comment with symbol * in front indicates a non-recursive goal whose execution dominates the complexity of a recursive step.

5.1. Time Complexity of the Original Rule

The time complexity of the original recursive rule is straightforward to derive.

Lemma 5.1. (Worst-Case Time Complexity of the Original Rule)

The worst-case time complexity $T_r(n)$ of taking n recursive steps with the given recursive rule $r = (H \Leftrightarrow C \mid D \wedge B \wedge H')$ can be derived from the recurrence equation

$$T_r(n) = T_b(n) + T_r(n-1),$$

where $T_b(n)$ is the time complexity of the n -th recursive step $C \wedge D \wedge B$ of rule r .

Proof. The recurrence follows directly from the structure of the linear direct recursive rule r . $\square$

5.2. Time Complexity of the Unfolder

For the unfolders we can derive the time complexity of its rules as follows:

```
unf(G, [R|Rs], URs) <=>           % head matching constant time
  R = (H <=> Co | _),               % unification constant time
  copy_term((H <=> Co), (G <=> C)), %*time linear in size of head and guard
  call(C)                           %*time of guard call of rule instance
  |
  simp_unf(R, UR),                  %*time for rule template computation
  unf(G, [UR,R|Rs], URs).          % recursive call

unf(_G, [_R|Rs], URs) <=> URs=Rs.  % matching and unification constant time
```

The complexity of the rule for the base case is constant. The complexity of a recursive step mainly depends on the time for copying head and guard, for guard checking, and for unfolding and simplification of the current rule.

Lemma 5.2. (Worst-Case Time Complexity of the Unfolder)

Given a terminating goal G that has recursion depth n with the given recursive rule r . Then the worst-case time complexity $T_{unf}(n)$ of the unfold `unf/3` for goal G with rule r can be derived from the the recurrence equation

$$T_{unf}(n) = T_c(n) + T_{unf}(n/2),$$

where

$$T_c(n) = c + T_{copy_term}(n) + T_{call_guard}(n) + T_{simp_unf}(n)$$

is the time complexity of a recursive step of the unfold with the unfolded rule r_i with $i = \lfloor \log_2(n) \rfloor$. In the summation, the notation T_p denotes the runtime of predicate p in the given code.

Proof. The base case of the unfold takes constant time and can therefore be ignored. The recurrence halves n . We show that this is correct. By Lemma 3.13 we know that k unfolded rules will be returned by the unfold such that $2^k \leq n$. In each recursive step, the unfold doubles the number of recursive steps covered by the currently unfolded rule and the number will not exceed n . Thus the complexity of generating these rules is the sum of $T_c(2^i)$ with $0 \leq i \leq k$. On the other hand, the recurrence halves n in each recursive step. This results in the sum of $T_c(n/2^j)$ with $0 \leq j \leq \log_2(n)$. But then for each $T_c(2^i)$ we have a corresponding $T_c(n/2^j)$ with $j = k - i$ such that $2^i \leq n/2^j$ since $2^k \leq n$. Therefore the recurrence provides an upper bound on the time complexity of the rules.

Finally, the definition of the complexity for the recursive step $T_c(n)$ can be directly read off the annotated code for the unfold given above. The constant c is the time needed for the head matching and the unification in the guard. $\square$

Note that the number of recursive steps of the unfold (and meta-interpreter) is logarithmic in the number of recursive steps of the original rule. This also reduces the overhead incurred by unfolding and meta-interpretation.

5.3. Time Complexity of the Meta-Interpreter

In the following code for the meta-interpreter, again comments indicate the runtime of each goal.

```
mip(true,_Rs) <=> true. % head matching and body execution constant time

mip(G,[R|Rs]) <=>      % head matching constant time
  copy_term(R, (G <=> C | B,G1,D)), %*time linear in rule size
  call(C)                %*runtime of guard C of rule instance
  |
  call(B),               %*runtime of goal B of rule instance
  mip(G1,Rs),            % recursive call
  call(D).               %*runtime of goal D of rule instance
```

```

mip(G,[_R|Rs]) <=>      % head matching constant time
    mip(G,Rs).           % recursive call

```

The second rule of the meta-interpreter applies a rule from the list to the current goal. It dominates the complexity. Its complexity is determined by the time needed for copying the rule and for the meta-calls of the guard and of the two body conjuncts of the rule. The third rule is also recursive in the rule list. The complexity of its recursive step is constant. The complexity of the rule for the base case is constant.

The resulting recurrence for complexity and its proof are analogous to the one for the unfold.

Lemma 5.3. (Worst-Case Time Complexity of the Meta-Interpreter)

Given a terminating goal G that has recursion depth n with the given recursive rule r . Then the worst-case time complexity $T_{mip}(n)$ of the meta-interpreter `mip/2` for goal G with rule r can be derived from the recurrence equation

$$T_{mip}(n) = T_d(n) + T_{mip}(n/2),$$

where

$$T_d(n) = c + T_{copy_term}(n) + T_{call_guard}(n) + T_{call_body}(n)$$

is the time complexity of a recursive step of the second rule of the meta-interpreter with the unfolded rule r_i with $i = \lfloor \log_2(n) \rfloor$. The runtime $T_{call_guard}(n)$ refers to the complexity of the goal `call(C)` for the guard and $T_{call_body}(n)$ to the complexity of the body goals `call(B)` and `call(D)`.

Proof. The base case of the meta-interpreter takes constant time and can therefore be ignored. The recurrence halves n . We show that this is correct. The unfold returned k unfolded rules with $2^k \leq n$ (cf. Lemma 3.13). These rules are ordered such that the more unfolded rules come first. In each recursive step, the meta-interpreter tries to apply the current unfolded rule once and then proceeds to the next one. Rule r_i covers 2^i recursive steps of the original rule r . Thus the complexity of applying these rules is the sum of $T_d(2^i)$ with $0 \leq i \leq k$.

The remainder of this proof is analogous to the one for the unfold: Since the recurrence halves n in each recursive step, it results in the sum of $T_d(n/2^j)$ with $0 \leq j \leq \log_2(n)$. But then for each $T_d(2^i)$ we have a corresponding $T_d(n/2^j)$ with $j = k - i$ such that $2^i \leq n/2^j$ since $2^k \leq n$. Therefore the recurrence provides an upper bound on the time complexity of the rules.

Again, the definition of the complexity for the recursive step $T_d(n)$ can be directly read off the annotated code for the meta-interpreter given above. The constant c is the time needed for the head matching. $\square$

Note that $T_d(n)$ has about the same time complexity as directly executing the rule (but possibly without optimizations), since the overhead of meta-calls is assumed to be constant and only the cost of copying the rule is added.

5.4. Time Complexity of Runtime Repeated Recursion Unfolding

We now can establish the worst-case time complexity of the recursive constraint under runtime repeated recursion unfolding. Recall that the original rules for the recursive constraint are replaced by the following rule that calls the unfold and then the meta-interpreter.

`rec_unfold @ G <=> unf(G, Rules, UnfoldedRules), mip(G, UnfoldedRules).`

Theorem 5.4. (Worst-Case Time Complexity of Runtime Repeated Recursion Unfolding)

Given runtime repeated recursion unfolding for the rules of a recursive constraint G and the time complexities $T_c(n)$ and $T_d(n)$ for a recursive step of the unfold and the meta-interpreter, respectively. Then the worst-case time complexity $T_u(n)$ of computing the recursion with runtime repeated recursion unfolding using the instance of rule `rec_unfold` for G can be derived from the recurrence equation

$$T_u(n) = T_c(n) + T_d(n) + T_u(n/2).$$

Proof. Clearly $T_u(n) = T_{unf}(n) + T_{mip}(n)$ according to rule `rec_unfold`. Recall the recurrence equations for the worst-case time complexity of the unfold and meta-interpreter:

$$\begin{aligned} T_{unf}(n) &= T_c(n) + T_{unf}(n/2) \text{ (cf. Lemma 5.2),} \\ T_{mip}(n) &= T_d(n) + T_{mip}(n/2) \text{ (cf. Lemma 5.3).} \end{aligned}$$

Hence $T_u(n) = (T_c(n) + T_{unf}(n/2)) + (T_d(n) + T_{mip}(n/2))$. We can replace $T_{unf}(n/2) + T_{mip}(n/2)$ by $T_u(n/2)$. Thus $T_u(n) = T_c(n) + T_d(n) + T_u(n/2)$. $\square$

We call $T_c(n) + T_d(n)$ the time complexity of the *combined recursive step* of the unfolded recursive constraint.

6. Super-Linear Speedup Theorems

We first define a class of time complexities. We then give general tight solutions for the recurrences for the recursions in terms of these complexities. In our speedup analysis, we then compare the time complexities of the recursive steps in the original recursion and in runtime repeated recursion unfolding. We establish relationships that lead to a super-linear speedup of the recursion. Based on them, we prove both a sufficient condition for super-linear speedup as well as a sufficient and necessary condition for super-linear speedup. For a given recursion, then one tries to find an unfolding and simplification with an improved time complexity that satisfies one of the conditions. If it can be found, a super-linear speedup is guaranteed.

In the following, we assume some familiarity with the Θ -notation for complexity and its manipulation (cf. Chapter 9.3 in [11] and Chapter 3 in [9]). The Θ -notation gives us an upper and lower bound on the complexity by ways of a complexity class.

6.1. Solving the Recurrences for Polylog-Polynomial Time Complexities

We consider time complexity classes that are expressible by polylog-polynomial functions of the form $n^j \log(n)^k$ in terms of recursion depth n where j and k are non-negative integers⁵. This includes as special cases polynomial complexity ($k = 0$), linear complexity ($k = 0, j = 1$), polylogarithmic complexity ($j = 0$), logarithmic complexity ($k = 1, j = 0$), and constant complexity ($k = 0, j = 0$).

⁵As we discuss in Section 9 this does not preclude exponential complexity in terms of problem size.

We can solve the recurrence without a boundary condition, i.e. without an extra equation for the base case of the recursion assuming the base case has constant time complexity (cf. Chapter 4 [9]).

Lemma 6.1. (Polylog-Polynomial Time Complexities in Runtime Repeated Recursion Unfolding)

Given a goal G with a recursive rule r that has recursion depth n and the recursive constraint resulting from runtime repeated recursion unfolding. Consider recursive steps and recursions with polylog-polynomial time complexities of the form $n^j \log(n)^k$ where $j \geq 0, k \geq 0$ are fixed non-negative integers.

Then for the original recursive rule r with time complexity $T_b(n)$ for the recursive step and time complexity $T_r(n)$ for the recursive computation it holds that

$$T_b(n) = \Theta(n^j \log(n)^k) \text{ iff } T_r(n) = \Theta(n^{j+1} \log(n)^k).$$

Then for runtime repeated recursion unfolding of rule r with time complexity $T_c(n) + T_d(n)$ for the combined recursive step and time complexity $T_u(n)$ for the recursive computation it holds that

$$T_c(n) + T_d(n) = \Theta(\log(n)^k) \text{ iff } T_u(n) = \Theta(\log(n)^{k+1}) \text{ and}$$

$$T_c(n) + T_d(n) = \Theta(n^j \log(n)^k) \text{ iff } T_u(n) = \Theta(n^j \log(n)^k) \text{ for } j \geq 1.$$

Proof. There are three claims. We first prove their implications to the right. We start with $T_b(n) = \Theta(n^j \log(n)^k) \Rightarrow T_r(n) = \Theta(n^{j+1} \log(n)^k)$ based on the recurrence for a recursive computation with the original rule from Lemma 5.1 $T_r(n) = T_b(n) + T_r(n-1)$. The complexity $\Theta(n^{j+1} \log(n)^k)$ is clearly an upper bound for $T_r(n)$ since there are n recursive steps and according to the claim $T_r(n) = n T_b(n)$. It remains to prove that the bound is tight, i.e. that it is also a lower bound. We compute a lower bound as follows: For the first $n/2$ recursive steps from n to $n/2$ we approximate $T_b(n)$ from below by $\Theta((n/2)^j \log_2(n/2)^k)$ and we ignore the contribution of the rest of the recursion. This gives a complexity of $\Theta((n/2)(n/2)^j \log_2(n/2)^k) = \Theta((n/2)^{j+1} (\log_2(n) - 1)^k) = \Theta(n^{j+1} \log_2(n)^k)$ for a fixed k . Hence the upper and lower bounds coincide.

The remaining two claims are based on the recurrence for runtime repeated recursion unfolding from Lemma 5.4 $T_u(n) = T_c(n) + T_d(n) + T_u(n/2)$. We next prove $T_c(n) + T_d(n) = \Theta(\log(n)^k) \Rightarrow T_u(n) = \Theta(\log(n)^{k+1})$. The complexity $\Theta(\log(n)^{k+1})$ is an upper bound for $T_u(n)$ since there are $\log_2(n)$ recursive steps and in the claim $T_r(n) = \Theta(\log_2(n) T_b(n))$. We prove that the bound is also a lower bound. For the first $\log_2(n)/2$ recursive steps starting from n we approximate $T_c(n) + T_d(n)$ from below by $\Theta((\log_2(n)/2)^k)$ and we ignore the contribution of the rest of the recursion. This gives a complexity of $\Theta((\log_2(n)/2)(\log_2(n)/2)^k) = \Theta((\log_2(n)/2)^{k+1}) = \Theta(\log_2(n)^{k+1})$ since k is fixed. Hence the upper and lower bounds coincide.

Now for $T_c(n) + T_d(n) = \Theta(n^j \log(n)^k) \Rightarrow T_u(n) = \Theta(n^j \log(n)^k)$ for $j \geq 1$. Since we have that $T_c(n) + T_d(n) = T_u(n)$ here, the complexity $\Theta(n^j \log(n)^k)$ is clearly a lower bound. We show the upper bound $T_u(n) \leq c n^j \log_2(n)^k$ with $n \geq 2$ for a suitably chosen constant $c \geq 1$. We will be using induction.

$$\begin{aligned} T_u(n) &= (T_c(n) + T_d(n)) + T_u(n/2) = n^j \log_2(n)^k + c (n/2)^j \log_2(n/2)^k \leq \\ &= n^j \log_2(n)^k + c (n^j / 2) \log_2(n)^k = n^j \log_2(n)^k (1 + c/2) \leq c n^j \log_2(n)^k \text{ if } c \geq 2. \end{aligned}$$

Time Complexity Class	Rec. Step $T_b(n)$	Orig. Rec. $T_r(n)$
polylog-polynomial $j \geq 0, k \geq 0$	$\Theta(n^j \log(n)^k)$	$\Theta(n^{j+1} \log(n)^k)$

Table 1. Polylog-Polynomial Time Complexity Classes of Original Recursion

Time Complexity Class	Rec. Step $T_c(n) + T_d(n)$	Unfold. Rec. $T_u(n)$
(poly)logarithmic, constant $j=0, k \geq 0$	$\Theta(\log(n)^k)$	$\Theta(\log(n)^{k+1})$
linear, (polylog-)polynomial $j \geq 1, k \geq 0$	$\Theta(n^j \log(n)^k)$	$\Theta(n^j \log(n)^k)$

Table 2. Polylog-Polynomial Time Complexity Classes of Runtime Repeated Recursion Unfolding

To prove the implications in the other direction for the three claims it suffices to observe that there is a bijective function (identity or increment) between the exponents of the complexity functions of the recursive steps and the recursions. Having proven one direction, it suffices to invert the functions to prove the other direction. $\square$

We summarize the results of the Lemma in Table 1 and Table 2. Note that for the original recursive rule, the time complexity always increases by a factor of $\Theta(n)$ when going from a recursive step to the complete recursion. For runtime repeated recursion unfolding, going from a recursive step to the complete recursion does not increase the worst-case time complexity for the classes that are at least linear, and by a factor of $\Theta(\log(n))$ for the polylogarithmic classes.

6.2. Sufficient Condition for Super-Linear Speedup

We have a *super-linear speedup* if the time complexity of runtime repeated recursion unfolding is lower than that of the original recursive computation, $\Theta(T_u(n)) \subset \Theta(T_r(n))$. The time complexities for the recursions depend on the time complexity for the respective recursive steps. Based on the general solutions for the recurrences we can now derive simple conditions on the complexity of the recursive steps that imply a super-linear speedup for the whole recursion. The idea then is to find a simplification of the recursive steps in the unfolded recursion that satisfies such a condition. If we succeed, a super-linear speedup is guaranteed.

We first consider a sufficient condition for super-linear speedup, where the combined recursive step of the unfold and of the meta-interpreter has the same time complexity as a recursive step with the original rule. Since the original recursive constraint takes n steps and the unfolded constraint just about $\log_2(n)$ steps, we expect a considerable speedup in that case.

Theorem 6.2. (Sufficient Condition for Super-Linear Speedup)

Given a goal with n recursive steps and time complexity $T_r(n)$ with the original recursive rule. Assume runtime repeated recursion unfolding with completeness of optimal rule applications (cf. Theorem 3.20) and recursive computations with time complexity $T_u(n)$ of the polylog-polynomial form $n^j \log(n)^k$ where j and k are fixed non-negative integers.

Time Complexity Class	Recursive Steps $T_b(n)=T_c(n)+T_d(n)$	Original Recursion $T_r(n)$	Repeatedly Unfolded $T_u(n)$
const., (poly)logarithmic $k \geq 0$	$\Theta(\log(n)^k)$	$\Theta(n \log(n)^k)$	$\Theta(\log(n)^{k+1})$
linear, polynomial, polylog-polynomial $j \geq 1, k \geq 0$	$\Theta(n^j \log(n)^k)$	$\Theta(n^{j+1} \log(n)^k)$	$\Theta(n^j \log(n)^k)$

Table 3. Time Complexity Classes for Super-linear Speedup with Sufficient Condition

Then we have a super-linear speedup

$$\Theta(T_u(n)) \subset \Theta(T_r(n)) \text{ if } \Theta(T_c(n) + T_d(n)) = \Theta(T_b(n)),$$

where $T_b(n)$ is the time complexity of the n -th recursive step with rule r .

Proof. Because $\log(n) < n$ where $n > c$ for some fixed constant c , the given condition $\Theta(T_c(n) + T_d(n)) = \Theta(T_b(n))$ implies $\log(n)\Theta(T_c(n) + T_d(n)) \subset n\Theta(T_b(n))$. By Lemma 6.1, we have that $\Theta(T_r(n)) = \Theta(n T_b(n))$ and the upper bound $\Theta(T_u(n)) \subseteq \Theta(\log(n)(T_c(n) + T_d(n)))$. Therefore $\log(n)\Theta(T_c(n) + T_d(n)) \subset n\Theta(T_b(n))$ implies $\Theta(T_u(n)) \subset \Theta(T_r(n))$. $\square$

Table 3 gives the complexities when this sufficient condition for super-linear speedup holds. For constant and polylogarithmic complexity classes, a super-linear speedup by the factor $\Theta(n/\log(n))$ is possible, and for the other polylog-polynomial time complexity classes, a super-linear speedup of $\Theta(n)$.

6.3. Sufficient and Necessary Condition for Super-Linear Speedup

Actually, we can already achieve a super-linear speedup if the complexity of the combined recursive step of the unfold and meta-interpreter is lower than that of *all* recursive steps (i.e. the complete recursion) with the original rule. We can even show that this conditions is not only sufficient, but also necessary.

Theorem 6.3. (Sufficient and Necessary Condition for Super-Linear Speedup)

Given a goal with n recursive steps with the original recursive rule. Assume runtime repeated recursion unfolding with completeness of optimal rule applications and recursions with polylog-polynomial time complexities.

Then we have a super-linear speedup

$$\Theta(T_u(n)) \subset \Theta(T_r(n)) \text{ iff } \Theta(T_c(n) + T_d(n)) \subset \Theta(n T_b(n)).$$

Proof. By Lemma 6.1, it holds that $\Theta(T_r(n)) = \Theta(n T_b(n))$. We know that $\Theta(T_c(n) + T_d(n))$ is of the form $n^j \log(n)^k$. By Lemma 6.1, it holds that $\Theta(T_u(n)) = \Theta(\log(n)(T_c(n) + T_d(n)))$ if $j = 0$ and $\Theta(T_u(n)) = \Theta((T_c(n) + T_d(n)))$ if $j \geq 1$. We consider these two cases.

If $j \geq 1$, then it directly follows from the equations in the Lemma that the two statements in our claim $\Theta(T_u(n)) \subset \Theta(T_r(n))$ and $\Theta(T_c(n) + T_d(n)) \subset \Theta(n T_b(n))$ are identical. If $j = 0$, then

Time Complexity Class	Rec.Step $T_b(n)$	Rec.Step $T_c(n)+T_d(n)$
constant, polynomial, linear $j \geq 0, k \geq 0$	$\Theta(n^j)$	$\Theta(n^j \log(n)^k)$
(poly)logarithmic, polylog-polynomial $j \geq 0, k \geq 1$	$\Theta(n^j \log(n)^k)$	$\Theta(n^{j+1} \log(n)^{k-1})$

Table 4. Highest Time Complexity Classes for Super-linear Speedup with Sufficient and Necessary Condition

it holds that $T_c(n) + T_d(n) = \log(n)^k$. So using the equations in the Lemma our claim becomes $\Theta(\log(n) \log(n)^k) \subset \Theta(n T_b(n))$ iff $\Theta(\log(n)^k) \subset \Theta(n T_b(n))$. Both sides hold since $\Theta(n) \subseteq \Theta(n T_b(n))$ and since the polylogarithmic complexity class is sub-linear, i.e. $\Theta(\log(n)^i) \subset \Theta(n)$ for any fixed i . $\square$

In Table 4 we list the highest complexities for a combined recursive step of the unfolded rules that still lead to a super-linear speedup. If the time complexity of the recursive step of the original recursion $T_b(n)$ is of the form $\Theta(n^j \log(n)^k)$ with $k \geq 1$, then the highest complexity class for the combined recursive step $T_c(n) + T_d(n)$ is $\Theta(n^{j+1} \log(n)^{k-1})$. If $T_b(n)$ is of the form $\Theta(n^j)$, then the highest complexity classes for the combined recursive step $T_c(n) + T_d(n)$ are of the form $\Theta(n^j \log(n)^k)$ for any fixed k . This follows from that fact that $\Theta(n^i \log(n)^h) \subset \Theta(n^j \log(n)^k)$ iff $[i, h] < [j, k]$ using lexicographic order⁶, because the polylogarithmic class is sub-linear.

7. Experimental Evaluation: Examples with Benchmarks

Our examples will demonstrate that super-linear speedups are indeed possible. With sufficient simplification, the time complexity is effectively reduced when applying runtime repeated recursion unfolding. In our experiments, we used the CHR library in SWI Prolog Version 6.2.1 running on an Apple Mac mini 2018 with Intel Core i5 8GB RAM and OS-X 10.14.6. We use default settings for SWI Prolog (including stack sizes) except for the command line option `-O` which compiles arithmetic expressions. During multiple runs of the benchmarks we observed a jitter in timings of at most 5%. Because the runtime improvement is so dramatic, we can only benchmark small inputs with the original recursion and have to benchmark larger inputs with runtime recursion unfolding.

7.1. Summation Example, Contd.

We have already unfolded and simplified the recursive rule for summation in Section 3, Example 3.4. We introduced the implementation in concrete syntax in Section 4.1, Example 4.1. We now derive estimates for the time complexities for our summation example and then compare them to benchmark results. We will predict and observe a super-linear speedup.

⁶See the more general case in Chapter 9.1, equation (9.6) in [11].

7.1.1. Complexity

Our example deals with arithmetic built-ins. SWI Prolog uses the GNU multiple precision arithmetic library (GMP), where integer arithmetic is unbounded. Comparison and addition have logarithmic worst-case time complexity in the numbers involved. Naive multiplication is quadratic in the logarithm. A variety of multiplication algorithms are used in GMP to get close to linear complexity. If one multiplies with a power of 2, the complexity can be reduced to logarithmic. This is the case in our example. We have confirmed this with some benchmarks in SWI Prolog.

Original Recursion The rule for the original recursion for summation is

$s(A,C) \text{ } \Leftarrow \text{ } A>1 \mid B \text{ is } A-1, s(B,D), C \text{ is } A+D.$

All numbers A, B, C and D are positive integers. By induction we can show that for a call $s(E, F)$ it holds that $(F/2)^2 \leq E \leq F^2$. The most costly arithmetic operation in the recursive step is the addition $C \text{ is } A+D$. The complexity of addition is logarithmic in its operands. The number D is the result of the recursive call $s(B, D)$. Hence D is quadratic in B and thus also quadratic in A , since $B \text{ is } A-1$. Since $A>1$, the recursion depth $n = A - 1$. So the time complexity of a recursive step $T_b(n)$ is the complexity for computing $C \text{ is } A+D$, which is $\Theta(\log(A) + \log(D)) = \Theta(\log(n) + \log(n^2)) = \Theta(\log(n) + 2\log(n)) = \Theta(\log(n))$. Hence the worst-case time complexity for the original recursive computation $T_r(n)$ is $\Theta(n \log(n))$ according to Lemma 6.1.

Unfolder By Lemma 5.2 the complexity of a recursive step of the unfolders can be derived from

$$T_c(n) = 1 + T_{\text{copy_term}}(n) + T_{\text{call_guard}}(n) + T_{\text{simp_unf}}(n).$$

Recall the predicate `simp_unf/2` for summation `s/2`

```
simp_unf(
  (s(A,C) <=> A>V | B is A-V, s(B,D), C is V*A-W+D),    % given rule template
  (s(A1,C1) <=> A1>V1 | B1 is A1-V1, s(B1,D1), C1 is V1*A1-W1+D1) % unfolded
) :-
  V1 is 2*V, W1 is 2*W+V*V.
```

Consider the definition of $T_c(n)$. For the complexity of $T_{\text{copy_term}}(n)$ and $T_{\text{call_guard}}(n)$ we observe the following: Copying head and guard of an unfolded summation rule and checking its guard involves the numbers A and V . Because of the guard $A>V$, the value of V is bounded by A , i.e. $n+1$. The size of an integer is logarithmic in its value. So $T_{\text{copy_term}}(n) = \Theta(2\log(A) + \log(V)) = \Theta(\log(n))$ and the comparison in the guard means that $T_{\text{call_guard}}(n) = \Theta(\log(A) + \log(V)) = \Theta(\log(n))$.

For the complexity $T_{\text{simp_unf}}(n)$ of `simp_unf/2`, consider the given rule template. The input is A and the parameters are V and W . All variables stand for positive integers. For the worst-case time complexity we need bounds on their values. We already know that C and D are quadratic in n and that A and V are bounded by $n+1$. So the product $V*A$ is bounded by $(n+1)^2$. Due to the computation $C \text{ is } V*A-W+D$, the parameter W is hence bounded by $2(n+1)^2$. The body of the clause for `simp_unf/2` contains $V1 \text{ is } 2*V$. Since the first value for V in the original recursion in template

form is 1, by induction V must be a power of 2. Overall, the clause body contains an addition and three multiplications that always involve a power of 2 (2 or V). So the time complexity of all arithmetic operations is logarithmic in the values involved. Since all values are positive, bounded by $2(n+1)^2$ and some values are quadratic in n , we arrive at a worst-case time complexity of $T_{simp_unf}(n) = \Theta(\log(2(n+1)^2)) = \Theta(\log(n))$.

Hence the time complexity for a recursive step of the unfold $T_c(n)$ is $\Theta(\log(n))$.

Meta-Interpreter Recall the complexity of a recursive step of the meta-interpreter according to Lemma 5.3

$$T_d(n) = 1 + T_{copy_term}(n) + T_{call_guard}(n) + T_{call_body}(n)$$

and recall that the template for unfolded summation rules is

$$s(A, C) \iff A > V \mid B \text{ is } A - V, \quad s(B, D), \quad C \text{ is } V * A - W + D.$$

As with the unfold, copying an unfolded summation rule can be done in logarithmic time. As for executing the guard and the non-recursive goals of the body, we have a comparison, subtractions, an addition and a multiplication in the rule. The multiplication is with V , a power of 2. All values of the variables involved are bounded by $2(n+1)^2$ and some are quadratic in n . So the time complexity for a recursive step of the meta-interpreter $T_d(n)$ is $\Theta(\log(n))$ as well.

Complexity of Runtime Repeated Recursion Unfolding The overall time complexity for a recursive computation with runtime repeated recursion unfolding $T_u(n)$ is $\Theta(\log(n)^2)$ according to Lemma 6.1. The complexity for $T_c(n)$ and $T_d(n)$ is the same as for a recursive step with the original rule $T_b(n)$, namely $\Theta(\log(n))$. We therefore satisfy the sufficient condition for super-linear speedup according to Theorem 6.2. So with repeated recursion unfolding the worst-case time complexity is reduced from $\Theta(n \log(n))$ to $\Theta(\log(n)^2)$.

7.1.2. Benchmarks

Table 5 shows benchmarks results for the summation example. Times are given in milliseconds. Experiments that show a runtime of less than 10 milliseconds are the averages of 1000 runs. The benchmarks confirm the super-linear speedup.

Original Recursion In each subsequent table entry, we double the input number. The runtime roughly doubles. So the runtime is at least linear. This is in line with the expected log-linear time complexity $\Theta(n \log(n))$: since the numbers are so small, addition is fast, almost constant time, and the runtime is dominated by the linear time overhead of the recursion itself. For larger numbers, the original recursion runs out of local stack.

Unfolder and Meta-Interpreter For runtime repeated recursion unfolding of our summation example, we give the time needed for the unfolding, the time needed for the execution with the meta-interpreter, and the sum of these timings (column 'Total Time'). Because our method has lower time

Original Summation		Runtime Repeated Recursion Unfolding			
Input n	Time	Input n	Unfolder	Interpreter	Total Time
2^{15}	3	2^{25}	0.03	0.03	0.06
2^{16}	6	2^{50}	0.07	0.08	0.15
2^{17}	12	2^{100}	0.18	0.18	0.36
2^{18}	24	2^{200}	0.41	0.40	0.81
2^{19}	48	2^{400}	0.84	0.82	1.66
2^{20}	108	2^{800}	1.80	1.72	3.52
2^{21}	217	2^{1600}	3.72	3.65	7.37
2^{22}	Out of stack	$2^{25} + 1$	0.03	0.02	0.05
		$2^{50} + 1$	0.07	0.05	0.12
		$2^{100} + 1$	0.18	0.10	0.28
		$2^{200} + 1$	0.40	0.19	0.59
		$2^{400} + 1$	0.84	0.39	1.23
		$2^{800} + 1$	1.76	0.80	2.56
		$2^{1600} + 1$	3.72	1.59	5.31

Table 5. Benchmarks for Summation Example (times in milliseconds)

complexity, it was already 5000 times faster than the original recursion for $n=2^{21}$. Hence we start from 2^{25} and in each subsequent table entry, we square the input number instead of just doubling it.

The runtimes of the unfolder and meta-interpreter are similar. For each squaring of the input number, the their runtimes more than double. The benchmarks results obtained are consistent with the expected complexity of $\Theta(\log(n)^2)$, e.g. $0.0000002 \log_2(n)^2 + 0.002 \log_2(n)$ for the unfolder.

Comparing Recursion Depths 2^i and $2^i + 1$ In the meta-interpreter, each of the unfolded rules will be tried by matching its head and checking its guard, but not all rules will be necessarily applied. This may lead to the seemingly counterintuitive behavior that a larger query runs faster than a smaller one.

Out of curiosity, to see how pronounced this phenomena is, we compare timings for values of n of the form 2^i and $2^i + 1$. Input numbers of the form $2^i + 1$ will need exactly one application of the most unfolded rule r_i to reach the base case, because the following recursive call has the input number computed by B is A-V which is $(2^i + 1) - 2^i$, i.e. 1. For numbers of the form 2^i however, all unfolded rules are applied. In this case, the most unfolded rule is r_{i-1} (not r_i), yielding a recursive call with input $2^i - 2^{i-1}$, i.e. 2^{i-1} . To this call, the next less unfolded rule r_{i-2} applies and so on. As a consequence it roughly halves the runtime of the meta-interpreter when going from a query with input number 2^i to $2^i + 1$. The timings for the unfolder stay about the same, because only one more rule is generated for $2^i + 1$ (e.g. $n = 2^{1600} + 1$ generates 1601 rules).

7.2. List Reversal Example

The classical program reverses a given list in a naive way. It takes the first element of the list, reverses its remainder and adds the element to the end of the reversed list. The CHR constraint $r(A, B)$ holds if list B is the reversal of list A .

$$\begin{aligned} r(E, D) &\Leftrightarrow E = [C|A] \mid r(A, B), a(B, [C], D) \\ r(E, D) &\Leftrightarrow E = [] \mid D = []. \end{aligned}$$

We use Prolog notation for lists. The term $[C|A]$ stands for a list with first element C and remaining list A . The built-in $a(X, Y, Z)$ appends (concatenates) two lists X and Y into a third list Z . Its runtime is linear in the length (number of elements) of the first list.

7.2.1. Runtime Repeated Recursion Unfolding

Our aim is to find the appropriate rule template for the repeated unfolding of the recursive rule with itself.

Unfolding We start with unfolding the original recursive rule with a copy of itself:

$$\begin{aligned} r(E, D) &\Leftrightarrow E = [C|A] \mid r(A, B), a(B, [C], D) \\ r(E', D') &\Leftrightarrow E' = [C'|A'] \mid r(A', B'), a(B', [C'], D'). \end{aligned}$$

The unfolding substitutes E' by A in the guard and produces

$$\begin{aligned} r(E, D) &\Leftrightarrow E = [C|A], A=[C'|A'] \mid r(A, B)=r(E', D'), r(A', B'), \\ &\quad a(B', [C'], D'), a(B, [C], D). \end{aligned}$$

This unfolding is correct because its three conditions are satisfied (cf. Def. 3.1). First, $r(A, B)$ is an instance of $r(E', D')$. The second condition requires $\text{vars}(A=[C'|A']) \cap \text{vars}(r(A, B)) \subseteq \text{vars}(r(E, D))$, i.e. $\{A\} \subseteq \text{vars}(r(E, D))$. This will hold if we consider the guard: since $r(E, D) \wedge E=[C|A] \equiv r([C|A], D) \wedge E=[C|A]$, we can replace E by $[C|A]$ and then $\{A\} \subseteq \text{vars}(r([C|A], D))$. Third and finally, the guard $E = [C|A], A = [C'|A']$ is satisfiable.

Simplification Now we proceed with rule simplification for unfolded rules (Definition 3.5). We simplify the head and guard by eliminating the local variable A .

$$r(E, D), E = [C|A], A=[C'|A'] \equiv_{\{E, D\}} r(E, D), E = [C, C'|A'].$$

For the body we first simplify by eliminating the local variables A, E' and D' .

$$\begin{aligned} E = [C|A], A=[C'|A'], r(A, B)=r(E', D'), r(A', B'), a(B', [C'], D'), a(B, [C], D) &\equiv_{\{E, D\}} \\ E = [C, C'|A'], r(A', B'), a(B', [C'], B), a(B, [C], D) & \end{aligned}$$

The insight for improving the time complexity is that we can merge the two calls to constraint $a/3$ into one if we concatenate their second arguments $[C']$ and $[C]$.

$$\begin{aligned} E &= [C, C'|A'], r(A', B'), a(B', [C'], B), a(B, [C], D) \equiv_{\{E, D\}} \\ E &= [C, C'|A'], r(A', B'), a(B', [C', C], D). \end{aligned}$$

Generalization The insight follows from the fact that list concatenation is associative. Consequently, we can simplify to two append constraints of the form $a(F, G, D), a(D, A, B)$, where the list G is sufficiently known, into $a(F, E, B)$, where E is the result of computing $a(G, A, E)$ already during simplification while unfolding.

This kind of simplification gives rise to a rule template of the following form

$$r(E, D) \Leftrightarrow E = [C_1, \dots, C_m|A'] \mid r(A', B'), a(B', [C_m, \dots, C_1], D).$$

We call $[C_1, \dots, C_m|A']$ an *open list*, because it ends in the list variable A' . The open list has size m because can match any list with at least m elements. The m elements $C_1, \dots, C_m$ are called *element variables*. Note that these element variables occur in reversed order in the list in the second argument of $a/3$ in the rule body.

7.2.2. Implementation

We use concrete syntax now and the built-in `append/3` for $a/3$.

Unfolding with Simplification The unfolding scheme for list reversal is implemented with the following Prolog clause for `simp_unf/2`.

```
simp_unf(
  (r(A,B) <=> A=E | true, r(C,D), append(D,F,B)),    % given rule template
  (r(A1,B1) <=> A1=E1 | true, r(C1,D1), append(D1,F1,B1)) % unfolded rule
) :-
  copy_term((E,C,F), (E1,Cc,Fc1)),
  copy_term((E,C,F), (Ec,C1,Fc2)),
  Cc=Ec,
  append(Fc2,Fc1,F1).
```

During unfolding, in the given rule template, the variable E in the guard will be instantiated with an open list ending in the variable C . The list F in `append/3` then consists of the element variables of E in reversed order. In the unfolded rule template, the number of elements in these two lists is doubled and their relationship of reversal is maintained.

The doubling is achieved by copying the guard list E together with its end variable C and list F twice. In the first copy, the guard list $E1$ ends in Cc . In the second copy, list Ec ends in $C1$ from the recursive call in the unfolded rule template. The variable Cc is unified with Ec from the second copy, thus doubling the number of element variables in $E1$. In this way, we have constructed a guard list $E1$ with twice as many element variables that ends in $C1$.

Finally, the lists resulting from copying F twice, $Fc1$ and $Fc2$, are concatenated in their reversed order by executing `append/3` in the body of the clause during unfolding. The result is the new reversed list $F1$ in `append/3` in the unfolded rule template.

Recursive Constraint For list reversal, the `rec_unfold` rule is as follows:

```
rec_unfold @ rev(I,0) <=>
  unf(r(I,0), [
    (r(A, E) <=> A=[D|B] | true, r(B, C), append(C, [D], E),
    (r(A, B) <=> A=[] | B=[], true, true)
  ], URs),
  mip(r(I,0), URs).
```

The list in the second argument of `unf/3` contains the original recursive rule and the rule for the base case in appropriate template form.

Unfolded Rules The rules that are returned by the unfolders `unf/3` for a query with 17 list elements are

```
r(A, T) <=> A=[S,R,Q,P,O,N,M,L,K,J,I,H,G,F,E,D|B] |
  true, r(B, C), append(C, [D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S], T).
r(A, L) <=> A=[K,J,I,H,G,F,E,D|B] |
  true, r(B, C), append(C, [D,E,F,G,H,I,J,K], L).
r(A, H) <=> A=[G,F,E,D|B] | true, r(B, C), append(C, [D,E,F,G], H).
r(A, F) <=> A=[E,D|B] | true, r(B, C), append(C, [D,E], F).
r(A, E) <=> A=[D|B] | true, r(B, C), append(C, [D], E).
r(A, B) <=> A=[] | B=[], true, true.
```

We see here an increase in *rule size*. With each unfolding, the rule size almost doubles because the number of elements in the lists double. For a query with n list elements, we unfold $\lfloor \log_2(n) \rfloor$ times. So the list in the most unfolded rule has not more than n elements. Therefore the size of all unfolded rules taken together will be proportional to n . Note that this does not increase overall space complexity, since the corresponding input list has n elements.

7.2.3. Complexity

We now derive estimates for the time complexities.

Original Recursion With the original rule we have n recursive steps for an input list of length n . The guard of the rule can be checked in constant time. In the body, `append/3` traverses the list in its first argument. The time needed is linear in the length of this list, which is n . So we have $T_b(n) = \Theta(n)$ for a recursive step. This results in the well-known quadratic complexity $\Theta(n^2)$ of naive list reversal.

Unfolder Recall the template for an unfolded rule of list reversal:

$r(A, B) \iff A=E \mid \text{true}, r(C, D), \text{append}(D, F, B).$

The sizes of the lists in the rule are bounded by the length n of the input list. We now consider a recursive step of the unfold. Copying head and guard of an unfolded rule as well as checking its guard has a runtime that is linear in the size of the open list E . In `simp_unf` we copy and concatenate the lists E and F . The worst-case time complexity of a recursive step in the unfold is therefore $T_c(n) = \Theta(n)$.

Meta-Interpreter In the meta-interpreter, copying an unfolded rule and checking its guard is linear in the size of the open list E . The time for concatenation with `append/3` is linear in the length of the list D . The runtime complexity of a recursive step in the meta-interpreter is therefore $T_d(n) = \Theta(n)$.

Complexity of Runtime Repeated Recursion Unfolding According to Lemma 6.1, this gives linear complexity $\Theta(n)$ in the input list length n for the unfold as well as the meta-interpreter and for both of them together. We therefore satisfy the sufficient condition for super-linear speedup according to Theorem 6.2. With repeated recursion unfolding the complexity is reduced from $\Theta(n^2)$ to $\Theta(n)$.

7.2.4. Benchmarks

Table 6 shows benchmarks results for the list reversal example. The list sizes n are powers of 2. Times are in seconds. A time measurement of 0.0n means that it was below 0.01 but more than zero. The experiments confirm the super-linear speedup using runtime repeated recursion unfolding.

Original Recursion For the original recursion, the benchmarks indicate a complexity consistent with the expected $\Theta(n^2)$. Doubling the list size increases the runtime by a factor of about four.

Unfolder and Meta-Interpreter All measured runtimes are consistent with a linear complexity $\Theta(n)$. For list size $n = 2^{13}$, runtime repeated recursion unfolding is already two orders of magnitude faster than the original recursion. A list with half a million elements can be reversed in half a second.

Comparing Recursion Depths $2^i - 1$ and 2^i To complete the picture, we give timings for list lengths n of the form 2^i and their predecessor numbers $2^i - 1$. In the meta-interpreter, the runtime of applying *all* unfolded rules (case of $n = 2^i - 1$) is less than of applying just the next larger unfolded rule (which has twice the size and complexity) (case of $n = 2^i$). The unfold takes several times longer than the meta-interpreter. Going from 2^{i-1} to 2^i , the unfold generates one more rule and the time spent doubles. Overall, going from $2^i - 1$ to 2^i almost doubles the total runtime.

Original list reversal		Runtime Repeated Recursion Unfolding			
Input n	Time	Input n	Unfolder	Interpreter	Total Time
2^9	0.01	$2^{13} - 1$	0.0n	0.0n	0.0n
2^{10}	0.04	$2^{14} - 1$	0.01	0.0n	0.01
2^{11}	0.16	$2^{15} - 1$	0.01	0.01	0.02
2^{12}	0.65	$2^{16} - 1$	0.02	0.01	0.03
2^{13}	2.88	$2^{17} - 1$	0.05	0.02	0.07
2^{14}	11.48	$2^{18} - 1$	0.09	0.04	0.13
2^{15}	46.36	$2^{19} - 1$	0.19	0.08	0.27
		2^{13}	0.01	0.0n	0.01
		2^{14}	0.01	0.0n	0.01
		2^{15}	0.02	0.01	0.03
		2^{16}	0.05	0.01	0.06
		2^{17}	0.09	0.03	0.12
		2^{18}	0.17	0.06	0.23
		2^{19}	0.36	0.14	0.50

Table 6. Benchmarks for List Reversal Example

7.3. Sorting Example

The classical insertion sort program sorts the numbers given in a list in ascending order:

$$s(L, S) \Leftrightarrow L = [A | L_1] \mid s(L_1, S_1), i(A, S_1, S)$$

$$s([], S) \Leftrightarrow S = [].$$

The built-in $i(A, S_1, S)$ inserts a number A into the sorted list S_1 such that the resulting list S is sorted.

7.3.1. Runtime Repeated Recursion Unfolding

Again we first have to find and define an appropriate rule template with sufficient simplification to improve on the runtime.

Unfolding Unfolding the recursive rule of $s/2$ results in the rule

$$s(L, S) \Leftrightarrow L = [A, A_1 | L_2] \mid s(L_2, S_2), i(A_1, S_2, S_1), i(A, S_1, S).$$

The number A_1 is inserted into the already sorted list S_2 , then into the resulting list S_1 , the number A is inserted. Repeating this unfolding scheme does not lead to any significant performance improvements, since we just generate more and more insertions.

Simplification The required simplification in this case is non-trivial. In the above rule, we observe that we can more efficiently insert both numbers A_1 and A during a single traversal of the list S_2 . We first insert the smaller number and then continue traversing the sorted list to insert the larger number. Since we get more and more insertions with each unfolding, we will actually have to insert more and more numbers in this way, and they have to be pre-sorted. To implement this behavior, we use a built-in $m(S_1, S_2, S_3)$ instead of insertions. It merges the sorted lists S_1 and S_2 into a sorted list S_3 .

In the above rule, we first order A and A_1 by putting them into a sorted list before they are merged with list S_2 . For the necessary ordering we will also use $m/3$. We replace the built-ins in the body of the rule $i(A_1, S_2, S_1), i(A, S_1, S)$ by the semantically equivalent $m([A], [A_1], S_0), m(S_0, S_2, S)$. The simplified unfolded rule for sorting is now

$$s(L, S) \Leftrightarrow L=[A, A_1|L_2] \mid m([A], [A_1], S_0), s(L_2, S_2), m(S_0, S_2, S).$$

The merging before the recursive call pre-sorts single numbers into a sorted list. The merging after the recursive call merges this list into the sorted list returned by the recursive call.

Now let us unfold this simplified rule with itself. The resulting rule is

$$s(L, S) \Leftrightarrow L=[A, A_1, A_2, A_3|L_3] \mid \\ m([A], [A_1], S_0), m([A_2], [A_3], S_1), s(L_3, S_3), m(S_1, S_3, S_2), m(S_0, S_2, S).$$

We now generate more and more mergings.

Generalization Note that after the recursive call, we merge the list of two elements S_1 into the already sorted list S_3 and the resulting list S_2 is in turn merged with the two elements of the list S_0 . We can improve the runtime if we rearrange the mergings so that we merge lists that are about the same length. We merge S_1 and S_0 first, move this merging before the recursive call and merge its result with S_3 after the recursive call:

$$s(L, S) \Leftrightarrow L=[A, A_1, A_2, A_3|L_3] \mid \\ m([A], [A_1], S_0), m([A_2], [A_3], S_1), m(S_1, S_0, S_4), s(L_3, S_3), m(S_4, S_3, S).$$

In this way we have almost halved the runtime by avoiding the generation and traversal of the intermediate sorted list S_2 .

The introduction of mergings is the essential idea for the simplification of the unfolded rules. It gives rise to the rule template

$$s(L, S) \Leftrightarrow L=[A, A_1, \dots, A_m|L_1] \mid \text{Mergings}, s(L_1, S_1), m(S_0, S_1, S).$$

The placeholder *Mergings* stands for the mergings of $A, A_1, \dots, A_m$ that result in the sorted list S_0 .

7.3.2. Implementation

We now implement the unfolding and the recursive constraint for sorting.

Unfolding with Simplification Relying on the rule template, the unfolding scheme is defined the following clause.

```
simp_unf(
  Rule,                                     % given rule
  (s(L,S) <=> L=AL | MG, s(L2,S2), m(S0,S2,S)) % unfolded rule template
  ):-
  copy_term(Rule, (s(L,S) <=> L=AL | MG1, s(L1,S1), m(S3,S1,S))),
  copy_term(Rule, (s(L1,S1) <=> L1=AL1 | MG2, s(L2,S2), m(S4,S2,S1))),
  clean((MG1, MG2, m(S3,S4,S0)), MG),
  L1=AL1.
```

We copy the input rule twice onto instances of the rule template to simulate the unfolding of the recursive call. In the first copy, the recursive call is $s(L1, S1)$. We directly use it as the head of the second copy of the given rule. The resulting unfolded rule is composed of the head of the first copy $s(L, S)$, of the guard of the first copy $L=AL$, of the mergings $MG1$ and $MG2$ before the recursive call of the two copies together with $m(S3, S4, S0)$, and the new merging after the recursive call $m(S0, S2, S)$. The built-in `clean/2` removes superfluous `true` constraints in the resulting mergings⁷. Finally, the resulting guard is completed by executing the guard of the second copy $L1=AL1$ at unfolding time. This will double the size of the open list AL which ends in $L1$.

Recursive Constraint For the sorting example, the `rec_unfold` rule is as follows:

```
rec_unfold @ sort(I,0) <=>
  unf(s(I,0), [
    (s(A,E) <=> A=[C|B] | true, s(B,D), m([C],D,E)),
    (s([],A) <=> true | A=[], true, true)
  ], URs),
  mip(s(I,0), URs).
```

We write the original recursive clause also in simplified form using `merge/3` instead of `insert/3`.

Unfolded Rules The first few rules that are returned by the unfolders with an appropriate query are

```
s(A,S) <=> A=[C,B,E,D,I,H,K,J|P] |
  ((m([B],[C],G), m([D],[E],F), m(F,G,O)),
   (m([H],[I],M), m([J],[K],L), m(L,M,N)), m(N,O,Q)),
  s(P,R), m(Q,R,S).
s(A,K) <=> A=[C,B,E,D|H] |
  (m([B],[C],G), m([D],[E],F), m(F,G,I)),
  s(H,J), m(I,J,K).
s(A,G) <=> A=[C,B|D] | m([B],[C],E), s(D,F), m(E,F,G).
s(A,E) <=> A=[C|B] | true, s(B,D), m([C],D,E).
s([],A) <=> true | A=[], true, true.
```

⁷The constraints stem from the original recursive clause and would proliferate otherwise.

As with list reversal, the rule size roughly doubles with each unfolding, but again this does not increase the space complexity.

7.3.3. Complexity

We derive estimates for the time complexities.

Original Recursion The recursion depth is determined by the number of elements n in the input list of the given query. In the original recursion we have n recursive steps. In each step, `insert/3` at worst traverses a list of length n . This results in the well-known quadratic complexity $\Theta(n^2)$ of insertion sort.

Unfolder Copying head and guard of an unfolded rule and checking its guard is linear in the size of the open input list. In `simp_unf/2` we basically copy the rule twice. The runtime complexity $T_c(n)$ of a recursive step in the unfolder is therefore linear in the input list length n , $\Theta(n)$.

Meta-Interpreter Copying an unfolded rule and checking its guard is linear in the size of the open input list. In the rule body, there are n calls to `merge/3` for an open input list of size n . These mergings dominate the complexity. The runtime of mergings is determined by the sum of the lengths of their input lists. The mergings of the singleton lists involve the n input list elements. The mergings of the resulting two-element lists also involve all n list elements. The mergings of all lists of the same length always involve all n input list elements. The lists double their lengths until all elements are merged before the recursive call. So we have a number of different list lengths that is logarithmic in n .

Overall, this results in a log-linear complexity for the mergings before the recursive call. After the recursive call, a list of length n is merged with the list resulting from the recursive call. The latter list cannot be larger than the former, because otherwise a more unfolded rule would have been applicable. In conclusion, the runtime complexity $T_d(n)$ of a recursive step in the meta-interpreter is therefore log-linear in the input list length n , $\Theta(n \log(n))$.

Complexity of Runtime Repeated Recursion Unfolding The solution of the associated recurrence equation in accordance with Lemma 6.1 maintains the log-linear complexity $\Theta(n \log(n))$ in the input list length n for the unfolder and the meta-interpreter together. We therefore satisfy the sufficient and necessary condition for super-linear speedup according to Theorem 6.3. Note that the unfolder itself has a lower, linear complexity. With repeated recursion unfolding the complexity is reduced from $\Theta(n^2)$ to $\Theta(n \log(n))$, clearly indicating a super-linear speedup.

7.3.4. Benchmarks

Table 7 shows benchmarks results for the sorting example. Times are in seconds. The benchmarks are performed with random permutations of integers from 1 to n . The individual runtimes show little variation, but are faster with already sorted lists, be they in ascending or descending order. They confirm the super-linear speedup.

Original Recursion The experiments for the original version of insertion sort indicate a complexity that is indeed quadratic $\Theta(n^2)$. Doubling the list length increases the runtime by a factor of four.

Unfolder and Meta-Interpreter The runtimes of the unfolder are consistent with a linear complexity $\Theta(n)$. The meta-interpreter timings are consistent with a log-linear complexity $\Theta(n \log(n))$. The generation of all rules in the unfolder takes less time than applying one or more rules in the meta-interpreter.

Comparing Recursion Depths $2^i - 1$ and 2^i Going from input list length $2^i - 1$ to 2^i , the unfolder generates one more rule. It has twice the size of the previous rule. And indeed the runtime for the unfolder almost doubles. Going from list length $2^i - 1$ to 2^i , the meta-interpreter applies all unfolded rules in the first case but only the next more unfolded rule in the second case. In both cases, all rules are tried by checking their guard. The runtime increases somewhat when going from $2^i - 1$ to 2^i .

Original Sorting		Runtime Repeated Recursion Unfolding			
Input n	Time	Input n	Unfolder	Interpreter	Total Time
2^9	0.01	$2^{12} - 1$	0.01	0.01	0.02
2^{10}	0.03	$2^{13} - 1$	0.01	0.02	0.03
2^{11}	0.13	$2^{14} - 1$	0.02	0.05	0.07
2^{12}	0.51	$2^{15} - 1$	0.04	0.11	0.15
2^{13}	2.20	$2^{16} - 1$	0.09	0.21	0.30
2^{14}	8.61	$2^{17} - 1$	0.19	0.47	0.66
2^{15}	34.24	$2^{18} - 1$	0.38	1.02	1.40
		$2^{19} - 1$	0.77	2.24	3.01
		2^{12}	0.01	0.01	0.02
		2^{13}	0.01	0.03	0.04
		2^{14}	0.04	0.06	0.10
		2^{15}	0.08	0.12	0.20
		2^{16}	0.16	0.27	0.43
		2^{17}	0.32	0.57	0.89
		2^{18}	0.65	1.34	1.99
		2^{19}	1.32	2.74	4.06

Table 7. Benchmarks for Sorting Example

8. Related Work

Program transformation to improve efficiency is usually concerned with a strategy for combining unfolding and folding to replace code (for an overview see e.g. [12, 13, 14]). The transformations are typically performed offline, at compile-time. Program transformation for specific aims and applications is abundant in logic programming in general [15] and in CHR in particular [16, 17]. General methods exist for unfolding [5] (which we have adapted for this paper), for specializing rules with regard to a specific given query [18], and for optimizations induced by confluence [19]. More recently, [20] uses program transformation implemented in CHR on constraint logic programs that verify properties of imperative programs.

Partial evaluation is a program transformation to execute programs with partially known input to specialize it, typically at compile-time. Simple partial evaluation alone cannot achieve super-linear speedup up (Chapter 6 in [21]). This linear speedup is called Type 1 speedup in [22]. This result does not apply to our approach, because we strongly rely on rule simplification. Involving an interpreter, our approach belongs to Type 2 speedup according to [22]. *Polyvariant program specialization* is the generation of specialized versions of a program according to different constraints that restrict its execution [23, 24]. One could argue that repeated recursion unfolding shares the same underlying idea, since it generates versions of a single rule specialized by recursion length.

In general, *super-linear speedups* by program transformation are rare and mostly concern parallel programs. Our technique applies to sequential programs. In a sequential setting, super-linear speedups can sometimes be achieved with *memoization*, where the results of recursive calls are cached and reused if the same recursive call reappears later on. A classical example where this runtime optimization applies is the naive double recursive implementation of the Fibonacci function. Typically, memoization pays off with multiple recursion, while our approach at the moment works with linear recursion. *Tupling* [25] is another technique that can achieve super-linear speedup. It applies when several recursions operate on the same data structure. Then tupling tries to merge these recursions into a single one. Memoization and tupling can be regarded as special cases of folding. Then there is work based on *supercompilation* for functional programming languages like Refal and Haskell. In advanced cases of this offline program transformation such as distillation [26] and equality indices [27], sophisticated generalization while unfolding increases the chance for folding and can achieve super-linear speedup on some examples. In contrast, our approach so far does not involve folding and works online at runtime. It requires a problem-specific simplification that has to be provided at compile-time.

The notion of *repeated recursion unfolding* was introduced in [1]. But there the rules were transformed at compile time. Because of this, super-linear speedup was only possible for calls that did not exceed a given fixed number of recursion steps. For larger calls, the speedup deteriorated to a constant factor. Here we substantially revised and greatly extended the approach for just-in-time (JIT) online execution. We introduced an unfolding scheme and a specialized meta-interpreter so that super-linear speedup can be made possible on-the-fly at runtime for any recursive call. Our technique relies solely on unfolding and simplifying the recursive step again and again. It ignores the base case of the recursion. We add redundant rules this way but never remove any. We never fold a recursive rule, but we simplify rule bodies (recursive steps).

As pointed out by a helpful reviewer, the basic idea of *repeatedly unfolding* a structural recursion

is sketched in work [28, 29] for Reform Prolog. In Prolog, there are no guards, so unfolding is rather straightforward. The unfolding in the cited work serves a different purpose, to parallelize the recursive steps. AND-parallelism requires that the recursive steps are (made) rather independent of each other, while in our approach it is essential to simplify the recursive steps for a sequential computation. This simplification benefits from dependence between the recursive steps. In this sense, the approach of Reform Prolog and of runtime repeated recursion unfolding are complementary: where simplification is not sufficient to achieve super-linear speedup, parallelization could be considered.

Compile-time *recursion unrolling* [30] for C inlines (unfolds) recursive calls, fuses (merges) conditionals and then re-rolls (folds) back the recursive part of the procedure to ensure a large simplified base case. The transformation is repeatedly applied, each time increasing the recursion depth by one. This technique is presented for double recursive divide and conquer algorithms where it can result in a constant factor speedup. In our approach, we work at runtime with linear recursion. We do not touch the base case at all and we do not fold. Repeated unfolding in our approach results in a doubling of the recursion depth covered, while in recursion unrolling, recursion depth is increased by one only. Conditional fusion merges only identical conditions, typically from the base case, while in our approach arbitrary guards are merged during unfolding. Recursion unrolling is cited mainly in work for parallelization and hardware programs, while we aim at sequential computations in software.

Recursion unrolling is derived from *loop unrolling* (Chapter 10.4.5 in [31], [32]) which is a standard code transformation in compilers that repeats the body of a loop a small fixed number of times. In this way, the overhead of the resulting code and the number of loop iterations can be reduced. On the other hand, prologue and epilogue code has to be added to account for the cases where the number of loop iterations is not a multiple of the iterations covered by the unrolled loop. The resulting speedup is a constant factor, typically less than two.

Finally, runtime repeated recursion unfolding should not be confused with *recursive doubling* [33, 34] (also called binary splitting in mathematics). In this optimization method, a problem is split top-down into two separate sub-problems of equal complexity that can be executed in parallel. Hence, a linear recursion would be transformed into a double recursion following the divide and conquer approach. In our method, we merge subsequent recursive steps and simplify them, doubling the number of recursive steps covered with each unfolding. No double recursion is introduced.

9. Discussion

We discuss some issues and limitations of runtime repeated recursion unfolding and suggest some possible improvements as well.

Rule Simplification. Our technique hinges on sufficient unfolding simplification of the recursive step resulting from unfolding. This simplification has to be provided at compile-time. It requires some insight into the given problem and cannot be fully automated (but mathematical software tools and theorem provers might help). Any existing optimization technique can be applied such as all kinds of program transformation. If the recursive steps are *arithmetic computations with polynomials*, they could be optimized using efficiently computable representations such as Horner’s method or more advanced approaches such as [35]. Another possibility is to use results from the verification of loops to compute closed forms (loop summarization [36], loop acceleration [37], loop solving [38]) for

sequences of recursive steps. While promising for arithmetic computations, this approach does not apply to structural recursion.

Often, simplification relies on *algebraic properties of the operations* in the recursive step. Judging from our experiments, we see that all examples involve the use of associativity of the operations to regroup the computation so that it becomes more efficient. For summation, it is the associativity of addition, for list reversal that of list concatenation, for sorting that of ordered merging. Another commonality is the standard optimization technique of finding *common sub-expressions*, i.e. to merge repeated data or operations. For summation, $N+N$ is replaced by $2*N$, for reversal and sorting, repeated traversals of lists are (partially) merged. To achieve this, we may need to replace operations, e.g. addition by multiplication for the summation example and insertion by merging for sorting.

Clearly, an algorithm implementation that is already optimal cannot be further improved. For a simple example, a search for the minimum of an unordered list has to go through all elements of the list. We cannot improve the time complexity of the linear direct recursion that performs this traversal without changing the data structure. A algorithm that keeps intermediate results of recursive steps can also be hard to optimize. For a simple example, this applies to a recursion that squares each number in a given list. But if the list contains successive integers, we can optimize the computations.

As one reviewer remarked, the necessary simplification scheme requires some effort, so one could go all the way deriving a more efficient algorithm for the recursion at hand. In particular, one could say our sort example is halfway towards deriving merge sort from insertion sort. On the other hand, the simplification in our list reversal example is not related to the common efficient version of reversal using an additional accumulator argument. The strength of our approach is that it provides a systematic practical and formally proven correct way to explore possible speedups and gives theorems when a super-linear speedup can be achieved.

Limited Recursion. Our approach as presented is restricted to single recursive rules. This does mean a loss of generality in terms of expressiveness. Any kind of recursion can be expressed as a linear recursion using continuation passing (such as in the rule-based language BinProlog [39]). The resulting linear recursive rules can be merged into a single such rule by introducing an auxiliary constraint performing the recursive steps. However, our preliminary experiments indicate that the resulting single linear recursive rule may be awkward and hard to optimize. Thus future work should consider multiple recursive rules directly. When insisting on the optimal rule application strategy, a naive extension of our approach could lead to a combinatorial explosion in the number of unfoldings.

Limited Time Complexity. We have considered complexity classes of the form $n^j \log(n)^k$, where the parameter n is the recursion depth. This allowed us to prove precise tight complexity results. However, complexity is usually stated in the size of the given problem. The size s often coincides with the recursion depth n (as was the case in our benchmarked problems), but it must not. For example, finding an element in a binary search tree by recursion has a complexity linear in the depth of the tree, but logarithmic in the size of the tree. This poses no problem, as our results carry over to complexity parametrized by problem size. To change the parameter, it suffices to find a non-constant positive monotonic function from size s to recursion depth n and replace n with it in the complexities. For the tree search, this gives $n = \log_2(s)$ and replacing n by $\log_2(s)$ leads to the correct complexity results. Exponential complexity in terms of size s is also possible and in this way a super-linear speedup into a polynomial complexity can be modeled. For upper bounds on complexity, it suffices

that the function limits n from above. Therefore our focus on tractable problems in terms of recursion depth does not preclude arbitrary complexities expressed in terms of problem size. In particular, our results also apply to exponential problems.

Space Complexity. Another issue are the space requirements of our approach. We generate a number of rules that is logarithmic in the recursion depth of the given query. In our examples we saw an increase in *rule size*. With each unfolding, the rule size roughly doubled. In effect, the size of all unfolded rules taken together is proportional to the size of the query, i.e. input number for the summation example and to length of the input list for reversal and sorting. Hence there was no increase space complexity. In general however, we cannot rule out code explosion in our approach.

Limited Unfolding. Rule unfolding in CHR has some conditions and may not be possible at all. Second, repeated unfolding may not produce enough rules to allow for optimal rule applications. So far, we have not observed these problems in practice. If they should occur, then we think they could be tackled with a more liberal definition of unfolding in CHR.

Possible Improvements. Note that unfolded rules are generic and can be reused for any later call, improving the efficiency further. As for the implementation, the following optimizations come to mind: The unfolders and the meta-interpreter can be specialized for a given recursive rule using standard partial evaluation techniques, which typically lead to an additional constant factor speedup. The unfolders and the meta-interpreter are currently head-recursive, the implementation could be made tail-recursive.

10. Conclusions and Future Work

We have introduced a strategy for online program optimization that is based on existing techniques such as unfolding that can achieve super-linear speedup. We have given a formal definition of runtime repeated recursion unfolding with simplification and proven its correctness. Our technique generates several versions of a single linear direct recursive rule for a recursive call at runtime, where each version doubles the number of recursive steps covered. The base case of the recursion is ignored. Our just-in-time method reduces the number of recursive rule applications to its logarithm at the cost of introducing a logarithmic number of unfolded rules. We provided a lean implementation of our approach in five rules, comprising the unfolders and the meta-interpreter and analyzed its complexity using recurrences. In our speedup analysis, we proved a sufficient condition as well as a sufficient and necessary condition for super-linear speedup relating the complexity of the recursive steps of the original rule and the unfolded rules. The results rely on an optimal rule application strategy that we proved sound and complete.

We showed with benchmarks on three simple basic algorithms that the super-linear speedup indeed holds in practice. For each example, we had to find a specific rule unfolding scheme. For ease of implementation, we used rule templates. Table 8 summarizes our estimated and observed time complexity results for our examples. They feature typical complexities of tractable algorithms and reduce the time complexity by a factor of $\Theta(n)$ or $\Theta(n/\log(n))$. For list reversal, the complexity of the given recursion was reduced to that of its recursive step. Summation and list reversal are examples for satisfying the sufficient condition for super-linear speedup. The sorting example does not, but satisfies the sufficient and necessary condition, with different complexities for the unfolders and meta-

Example	R.Step $T_b(n)$	Recursion $T_r(n)$	R.Step $T_c(n)+T_d(n)$	Unfolded $T_u(n)$
Summation	$\Theta(\log(n))$	$\Theta(n \log(n))$	$\Theta(\log(n) + \log(n))$	$\Theta(\log(n)^2)$
List Reversal	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n + n)$	$\Theta(n)$
Sorting	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n + n \log(n))$	$\Theta(n \log(n))$

Table 8. Summary: Time Complexity Classes of Super-linear Speedup for Examples

interpreter.

Overall, runtime repeated recursion unfolding provides a general strategy for online optimization of linear direct recursions in which the sufficient simplification of successive recursive steps leads to predictable speedups.

Future work. This paper introduces our approach, but does not explore it in full. Our main limitation is the challenge of finding sufficient problem-specific simplifications. Future work should investigate classes of functions that can be simplified in the necessary way, such as polynomial arithmetic expressions. We assumed a single recursive rule with linear direct recursion written in CHR. As we have discussed, this does not result in a loss of generality in terms of expressiveness. However, this restriction may lead to unnatural implementations that are hard to optimize. Hence we want to extend our technique to mutual and multiple recursion as well as multiple recursive rules.

We defined and implemented repeated recursion unfolding using the rule-based language CHR, but we think our approach can be applied to other rule-based languages and mainstream programming languages as well. First candidates are other declarative programming languages like Prolog and Haskell. In Prolog we will have to deal with non-determinism in the rule choice, in functional languages we will have the issue of nested guards and conditionals. For the implementation, meta-programming features may not be necessary if the interpreter is specialized with regard to the given recursion so that the meta-calls go away. It already might be an advantage that the number of recursive steps is reduced to its logarithm by our approach. For example, there is a limit on recursion depth in languages like Java and Python due to the limit on stack size. Last but not least, it should also be possible to apply our technique to loops instead of recursion.

Acknowledgements. This research work was initiated during the sabbatical of the author in the summer semester of 2020. We thank the anonymous reviewers and Sascha Rechenberger for comments. In particular, one reviewer made highly detailed and substantial comments and suggestions that helped tremendously to clarify, improve and extend the paper.

References

- [1] Frühwirth T. Repeated Recursion Unfolding for Super-Linear Speedup within Bounds. *Pre-Proceedings of the 30th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2020)*, full version arXiv preprint arXiv:2009.05314, 2020.
- [2] Frühwirth T. Constraint Handling Rules. Cambridge University Press, 2009. ISBN 9780521877763.

- [3] Raiser F, Betz H, Frühwirth T. Equivalence of CHR States Revisited. In: Raiser F, Sneyers J (eds.), CHR '09. K.U.Leuven, Dept. Comp. Sc., Technical report CW 555, 2009 pp. 33–48.
- [4] Betz H. A unified analytical foundation for constraint handling rules. BoD, 2014.
- [5] Gabbriellini M, Meo MC, Tacchella P, Wiklicky H. Unfolding for CHR programs. *Theory and Practice of Logic Programming*, 2015. **15**(3):264–311.
- [6] Duck GJ, Stuckey PJ, García de la Banda M, Holzbaur C. The Refined Operational Semantics of Constraint Handling Rules. In: Demoen B, Lifschitz V (eds.), ICLP '04, volume 3132 of *LNCS*. Springer. ISBN 978-3-540-22671-0, 2004 pp. 90–104. doi:10.1007/b99475.
- [7] Wielemaker J, Schrijvers T, Triska M, Lager T. SWI-Prolog. *Theory and Practice of Logic Programming*, 2012. **12**(1-2):67–96.
- [8] Schrijvers T, Wielemaker J, Demoen B. Constraint Handling Rules for SWI-prolog. *WCLP*, 2005. **5**:2005–01.
- [9] Thomas H C, Charles E, Ronald L R, Clifford S. Introduction to Algorithms, Third Edition. MIT Press, 2009.
- [10] Frühwirth T. As time goes by II: More automatic complexity analysis of concurrent rule programs. *Electronic Notes in Theoretical Computer Science*, 2002. **59**(3):185–206.
- [11] Graham RL, Knuth DE, Patashnik O. Concrete Mathematics: A Foundation for Computer Science. Addison-Wesley, Reading, MA, second edition, 1994. ISBN 0201558025 9780201558029 0201580438 9780201580433 0201142368 9780201142365.
- [12] Pettorossi A, Proietti M. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys (CSUR)*, 1996. **28**(2):360–414.
- [13] Visser E. A survey of strategies in rule-based program transformation systems. *Journal of symbolic computation*, 2005. **40**(1):831–873.
- [14] Pettorossi A, Proietti M, Fioravanti F, De Angelis E. A Historical Perspective on Program Transformation and Recent Developments (Invited Contribution). In: Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation. 2024 pp. 16–38.
- [15] Pettorossi A, Proietti M. Synthesis and transformation of logic programs using unfold/fold proofs. *The Journal of Logic Programming*, 1999. **41**(2-3):197–230.
- [16] Sneyers J, Van Weert P, Schrijvers T, De Koninck L. As Time Goes By: Constraint Handling Rules – A Survey of CHR Research between 1998 and 2007. *TPLP*, 2010. **10**(1):1–47. doi:10.1017/S1471068409990123.
- [17] Frühwirth T. Constraint Handling Rules - What Else? In: Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings. 2015 pp. 13–34. doi:10.1007/978-3-319-21542-6_2. URL https://doi.org/10.1007/978-3-319-21542-6_2.
- [18] Frühwirth T. Specialization of Concurrent Guarded Multi-Set Transformation Rules. In: Etalle S (ed.), LOPSTR '04, volume 3573 of *LNCS*. Springer, 2005 pp. 133–148.
- [19] Abdennadher S, Frühwirth T. Integration and Optimization of Rule-based Constraint Solvers. In: Bruynooghe M (ed.), LOPSTR '03, volume 3018 of *LNCS*. Springer, 2004 pp. 198–213.

- [20] De Angelis E, Fioravanti F, Pettorossi A, Proietti M, Giordano L, Gliozzi V, Pettorossi A, Pozzato GL. Program Verification Using Constraint Handling Rules and Array Constraint Generalizations. *Fundamenta Informaticae*, 2017. **150**(1):73–117. doi:10.3233/FI-2017-1461. URL <https://doi.org/10.3233/FI-2017-1461>.
- [21] Jones ND, Gomard CK, Sestoft P. Partial evaluation and automatic program generation. Prentice Hall international series in computer science. Prentice Hall, 1993. ISBN 978-0-13-020249-9.
- [22] Jones ND. Transformation by interpreter specialisation. *Sci. Comput. Program.*, 2004. **52**:307–339. doi:10.1016/j.scico.2004.03.010. URL <https://doi.org/10.1016/j.scico.2004.03.010>.
- [23] Angelis ED, Fioravanti F, Gallagher JP, Hermenegildo MV, Pettorossi A, Proietti M. Analysis and Transformation of Constrained Horn Clauses for Program Verification. *Theory and Practice of Logic Programming*, 2022. **22**(6):974–1042. doi:10.1017/S1471068421000211.
- [24] Gallagher JP. Polyvariant program specialisation with property-based abstraction. In: Seventh International Workshop on Verification and Program Transformation (VPT 2019). EPTCS 299, 2019 pp. 34–48.
- [25] Hu Z, Iwasaki H, Takeichi M, Takano A. Tupling calculation eliminates multiple data traversals. *ACM Sigplan Notices*, 1997. **32**(8):164–175.
- [26] Hamilton GW. Extracting the essence of distillation. In: International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Springer, 2009 pp. 151–164.
- [27] Glück R, Klimov A, Nepeivoda A. Nonlinear Configurations for Superlinear Speedup by Supercompilation. In: Fifth International Valentin Turchin Workshop on Metacomputation. University of Pereslavl, 2016 p. 32.
- [28] Millroth H. Using the Reform inference system for parallel Prolog. In: Fronhöfer B, Wrightson G (eds.), Parallelization in Inference Systems. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-47066-3, 1992 pp. 182–194.
- [29] Millroth H. SLDR-resolution: Parallelizing structural recursion in logic programs. *The Journal of Logic Programming*, 1995. **25**(2):93–117. doi:[https://doi.org/10.1016/0743-1066\(95\)00036-J](https://doi.org/10.1016/0743-1066(95)00036-J). URL <https://www.sciencedirect.com/science/article/pii/074310669500036J>.
- [30] Rugina R, Rinard MC. Recursion Unrolling for Divide and Conquer Programs. In: Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers, LCPC '00. Springer-Verlag, London, UK, UK. ISBN 3-540-42862-3, 2001 pp. 34–48. URL <http://dl.acm.org/citation.cfm?id=645678.663942>.
- [31] Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: Principles, Technologies, and Tools. Addison Wesley, 2006.
- [32] Leopoldseder D, Schatz R, Stadler L, Rigger M, Würthinger T, Mössenböck H. Fast-path loop unrolling of non-counted loops to enable subsequent compiler optimizations. In: Proceedings of the 15th International Conference on Managed Languages & Runtimes. 2018 pp. 1–13.
- [33] Kogge PM, Stone HS. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers*, 1973. **100**(8):786–793.
- [34] Afrati FN, Ullman JD. Transitive closure and recursive datalog implemented on clusters. In: Proceedings of the 15th International Conference on Extending Database Technology. 2012 pp. 132–143.

- [35] Leiserson CE, Li L, Maza MM, Xie Y. Efficient Evaluation of Large Polynomials. In: Fukuda K, Hoeven Jvd, Joswig M, Takayama N (eds.), *Mathematical Software – ICMS 2010*. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-15582-6, 2010 pp. 342–353.
- [36] Kincaid Z, Breck J, Cyphert J, Reps T. Closed Forms for Numerical Loops. *Proc. ACM Program. Lang.*, 2019. **3**(POPL). doi:10.1145/3290368. URL <https://doi.org/10.1145/3290368>.
- [37] Frohn F. A calculus for modular loop acceleration. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020 pp. 58–76.
- [38] Kafle B, Gallagher JP, Hermenegildo MV, Klemen M, López-García P, Morales JF. Regular Path Clauses and Their Application in Solving Loops. *Electronic Proceedings in Theoretical Computer Science*, 2021. **344**:22–35. doi:10.4204/eptcs.344.3. URL <http://dx.doi.org/10.4204/EPTCS.344.3>.
- [39] Tarau P. The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. *Theory and Practice of Logic Programming*, 2012. **12**(1-2):97–126.