Repeated Recursion Unfolding for Super-Linear Speedup within Bounds

Thom Frühwirth

University of Ulm, Germany

LOPSTR 2020

Repeated Recursion Unfolding

- *Repeated Recursion Unfolding* repeatedly **unfolds** a recursion with itself and **simplifies** it while keeping all unfolded rules.
- Each unfolding doubles the number of recursive steps covered.
  Best-case simplification keeps runtime bounded.
- **Super-linear speedup in the best case up to a chosen bound.**
- In an implementation, we remove recursion up to the chosen bound.
- Runtime improvement quickly reaches several orders of magnitude.

Example Summation

We define and implement our approach in Constraint Handling Rules (CHR).

Example (Summation)

Add all numbers from 1 to n.

$$sum(N,S) \Leftrightarrow N > 1 \mid S := N+S1, sum(N-1,S1)$$
$$sum(N,S) \Leftrightarrow N = 1 \mid S = 1$$

We never unfold with the base case. It is ignored.

Example Summation

We define and implement our approach in Constraint Handling Rules (CHR).

Example (Summation)

Add all numbers from 1 to n.

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

$$sum(N,S) \Leftrightarrow N > 1 \mid S := N{+}S1, sum(N{-}1, S1)$$

$$sum(N,S) \Leftrightarrow N = 1 \mid S = 1$$

We never unfold with the base case. It is ignored.

Example Summation

We define and implement our approach in Constraint Handling Rules (CHR).

Example (Summation)

Add all numbers from 1 to n.

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

We never unfold with the base case. It is ignored.

Example Summation Recursively Unfolded

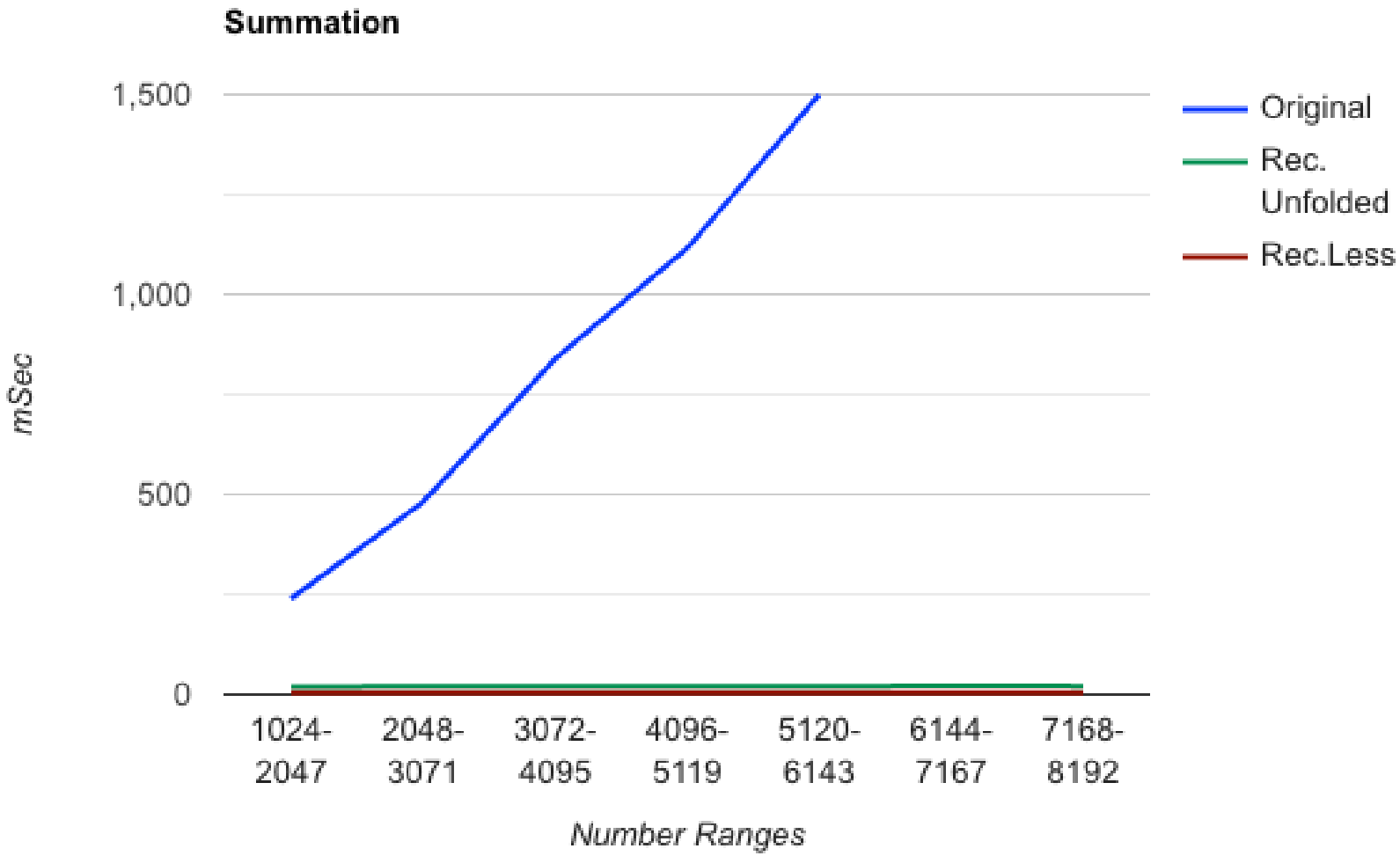
Example (Summation, contd.)

$$sum(N,S) \Leftrightarrow N > 8 \,|\, S := 8 * N - 28 + S1, sum(N-8, S1)$$
$$sum(N,S) \Leftrightarrow N > 4 \,|\, S := 4 * N - 6 + S1, sum(N-4, S1)$$
$$sum(N,S) \Leftrightarrow N > 2 \,|\, S := 2 * N - 1 + S1, sum(N-2, S1)$$
$$sum(N,S) \Leftrightarrow N > 1 \,|\, S := N + S1, sum(N-1, S1)$$
$$sum(N,S) \Leftrightarrow N = 1 \,|\, S = 1$$

Apply the most unfolded rule possible in each recursive step (rule order).

Example Summation Benchmarks

Repeated Recursion Unfolding improves runtime from linear to constant.



Rule Unfolding

Definition (Unfolding)

[Gabrielli et.al. 2015] Given two rules

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

where S matches H', i.e. $S{=}H'\theta$, then

$$unfold(r, v) = r' : H \Leftrightarrow C \wedge C''\theta \,|\, D \wedge B \wedge S{=}H' \wedge B',$$

where $C''\theta$ is $C'\theta$ with constraints also in C and D removed.

Proven correct if the variables shared between $H'\theta$ and $C''\theta$ also occur in H.

Rule Simplification

Definition (Simplification)

Given a rule

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

where D are the built-in constraints, then

$$simplify(r) = (H' \Leftrightarrow C' \,|\, D'' \wedge B') \text{ such that}$$
$$(H \wedge C) \equiv (H' \wedge C') \text{ and } (C \wedge D \wedge B) \equiv (D' \wedge B'),$$

where D'' is D' with constraints already in C' removed.
Relation $\equiv$ denotes equivalence between constraints.

Proven correct: the simplified rule behaves like the original rule.

Repeated Recursion Unfolding

Definition (Repeated Recursion Unfolding)

The *unfolding of a recursive rule* r is

$$unfold(r) = unfold(r, r)$$

The *repeated unfolding* is a sequence of rules $r_0, r_1, \ldots, r_i, \ldots$ where

$$r_0 = r$$

$$r_{i+1} = simplify(unfold(r_i))$$

Let n be an upper bound on the number of recursive steps (recursion depth).
The *recursively unfolded program* is

$$\mathcal{P}^{r,n} = \mathcal{P} \cup \bigcup_{i=1}^{\lfloor \log_2(n) \rfloor} r_i$$

Optimal Rule Applications

Lemma (Optimal Rule Applications)

Given unfolded program $\mathcal{P}^{r,n}$.
IF
If a rule r_i can perform two recursive computation steps, then rule r_{i+1} can perform one computation step with the same result.
THEN
If original rule r takes n recursion steps, then we can do with at most $log_2(n)$ rule applications by always applying the most unfolded rule possible.

We can reduce the number of recursive rule applications to its logarithm at the expense of introducing a logarithmic number of unfolded rules to the program.

Super-Linear Speedup Theorem

Definition

Function $c(n)$ computes a time bound for the first recursive step with any recursive rule r_i with $i \leq log_2(n)$ for any query with recursion depth n.
Best-case simplification: unfolded rules have the same time bound $c(n)$ as original rule.

Theorem (Super-Linear Speedup of Repeated Recursion Unfolding)

Given $\mathcal{P}^{r,n}$ *with optimal rule applications and best-case simplification, THEN*

Time Complexity Class	*Rec.Step* $c(n)$	*Rec.*	*Unfolded*
(poly)logarithmic, constant $k \geq 0$	$log_2(n)^k$	$nlog_2(n)^k$	$log_2(n)^{k+1}$
polynomial, linear $k \geq 1$	n^k	n^{k+1}	$2n^k$

Recursionless Recursion

Semi-naive implementation of optimal rule applications that removes recursion.

Definition (Recursionless Recursion)

Replace each rule r_i with recursive call $R = r(\ldots)$

$$r_i : H \Leftrightarrow C \,|\, D \wedge B \wedge R$$

by the pair of rules

$$r_i : H_i \Leftrightarrow C \,|\, D \wedge B \wedge R_{i-1}$$
$$r'_i : r(\bar{x})_i \Leftrightarrow r(\bar{x})_{i-1}$$

where $\bar{x}$ are disjoint variables as arguments.

Super-Linear Speedup with Rule Order and Recursionless Recursion

Rule Order means to try rules in order given in the program.

Lemma (Speedup with Worst-Case Overhead)

Let $N > n > 1$. *Repeatedly unfold for recursion depth* N.
THEN
The worst-case slow-down over optimal rule applications is linear in the number of unfolded rules $log_2(N)$.

Example List Reversal

Example (List Reversal)

The original recursion reverses a given list in a naive way.

$$r([C|A], D) \Leftrightarrow r(A, B), a(B, [C], D)$$
$$r([], D) \Leftrightarrow D = []$$

The built-in constraint $a(X, Y, Z)$ appends two lists X and Y.

Example List Reversal

Example (List Reversal)

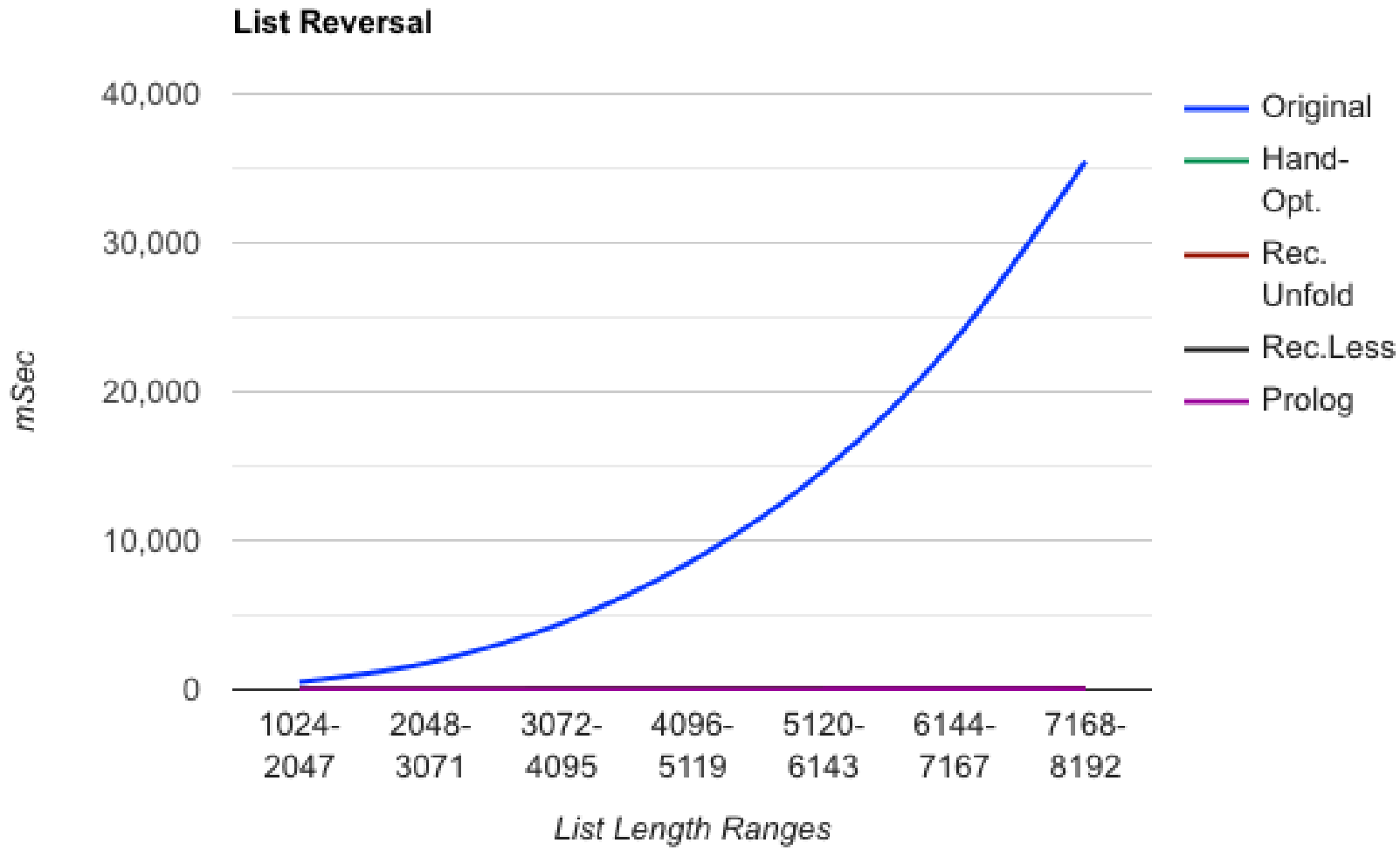
The original recursion reverses a given list in a naive way.

$$r([J,I,H,G,F,E,D,C|A],K) \Leftrightarrow r(A,B), a(B,[C,D,E,F,G,H,I,J],K)$$
$$r([F,E,D,C|A],G) \Leftrightarrow r(A,B), a(B,[C,D,E,F],G)$$
$$r([D,C|A],E) \Leftrightarrow r(A,B), a(B,[C,D],E)$$
$$r([C|A],D) \Leftrightarrow r(A,B), a(B,[C],D)$$
$$r([],D) \Leftrightarrow D = []$$

The built-in constraint $a(X,Y,Z)$ appends two lists X and Y.

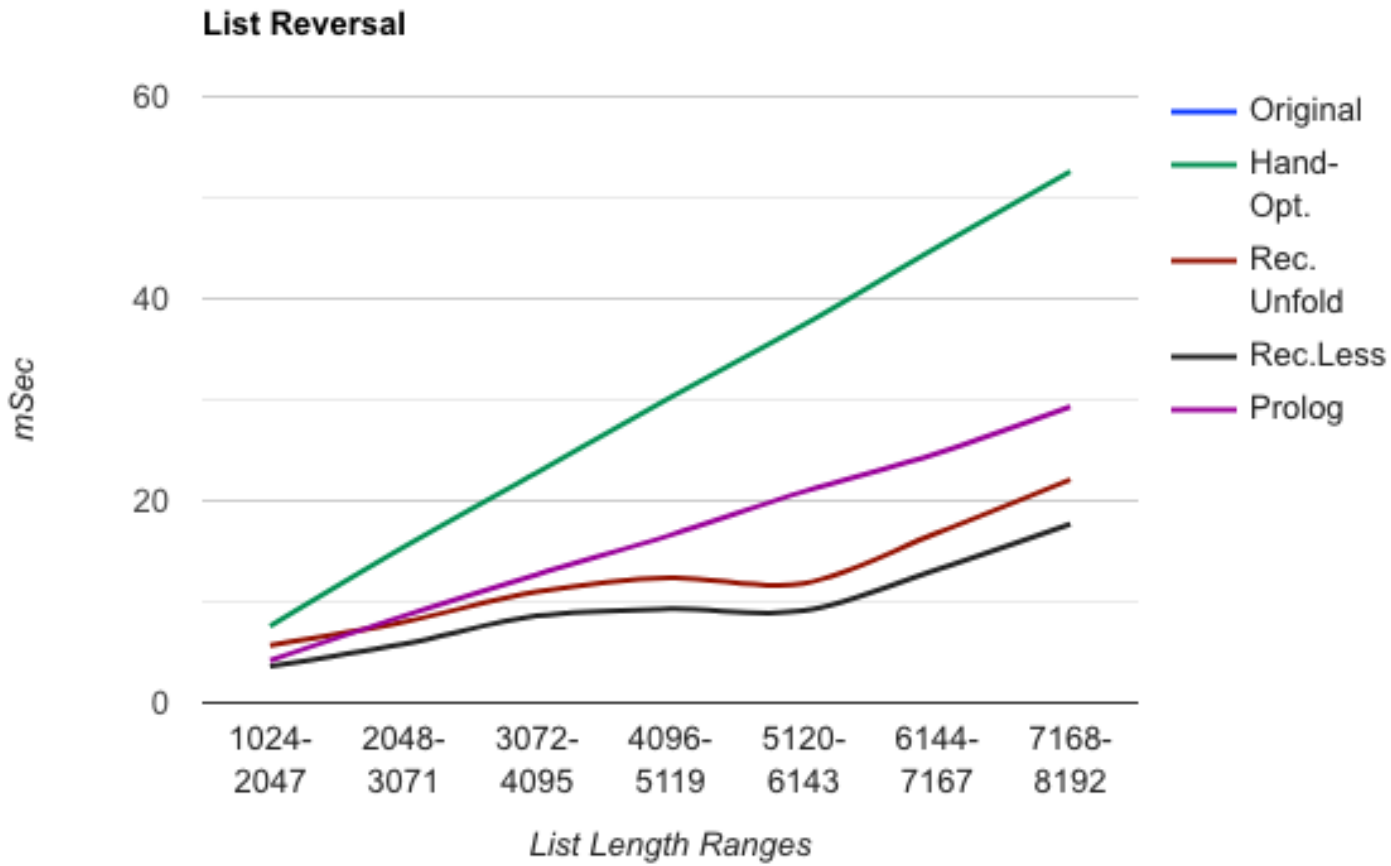
Example List Reversal Benchmarks

Repeated Recursion Unfolding improves runtime from quadratic to linear.



Example List Reversal Benchmarks

Zoom previous chart by two orders of magnitude:



Twice as fast as the hand-optimized and faster than built-in reversal.

Related Work

- **Program transformations** for efficiency use unfolding and folding to replace code. Recursion typically unfolded with base case.
  BUT Repeated Recursion Unfolding only generates and keeps redundant recursive rules. It ignores the base case.
- **Super-linear speedups** are rare and mostly for parallel programs.
  BUT Repeated Recursion Unfolding applies to sequential programs.
- **Exception:** *supercompilation extended with generalisation.*
  BUT Repeated Recursion Unfolding is without generalisation and folding.
- **Related:** *Unfolding-based meta-level systems* [Amtoft 1991] for Prolog consist of a hierarchy of meta-rules and a hierarchical execution scheme.

Conclusions

- **Repeated Recursion Unfolding** unfolds a recursion with itself and simplifies it while keeping all unfolded rules up to a given bound.
- We proved a **super-linear speedup theorem** in case of best-case simplification.
- *Rule Order* and *Recursionless Recursion:* **semi-naive implementations sufficient for super-linear speedup.** Runtime improvement quickly reaches several orders of magnitude.
- *Best-case simplification* **requires some insight** and is not always possible.

Ongoing and Future Work

- Use **indexing** on recursion depth for optimal rule applications.
  DONE for the examples in this paper.
- Extend to **double and mutual recursion** as well as multiple recursive rules.
  DONE linear-time double recursion from exponential-time Fibonacci.
- Go for **unbounded super-linear speedup**: run-time dynamic on-the-fly just-in-time Repeated Recursion Unfolding.
  DONE for the examples in this paper.
- Transfer our approach to **other programming languages** and from recursion to **loop constructs.**
  DONE for examples in Java.

Acknowledgements

- Part of this research work was performed during the sabbatical of the author in summer semester 2020.
- We thank the anonymous reviewers for their skepticism which helped to clarify the contribution of the paper.