ReCheck: Automated Contextual Improvement Verifier for Functional Calculi across User-Defined Operational Semantics

Makoto Hamana$^{1[0000-0002-3064-8225]}$ and Kento Emoto$^{1[0000-0002-7608-0065]}$

Department of Computer Science and Networks, Kyushu Institute of Technology
{hamana,emoto}@csn.kyutech.ac.jp

Abstract. Contextual improvement is a directed form of contextual equivalence that ensures a reduction in the number of evaluation steps. We present ReCheck, an automated tool that verifies contextual improvement for foundational calculi (such as extensions of the λ-calculus) and higher-order functional programs across operational semantics defined by Felleisen's evaluation contexts. ReCheck allows users to define syntax, evaluation, and refinement rules as rewriting systems called Term Evaluation and Refinement Systems (TERS). We implement the theory of TERS and the method for deriving contextual improvement proposed by Muroya and Hamana in the tool ReCheck. We demonstrate ReCheck's applicability by proving contextual improvement across diverse calculi and programs.

1 Introduction

Contextual equivalence [23] is a fundamental concept in programming language theory, capturing the idea that two program fragments are indistinguishable in all contexts under a given operational semantics. Contextual improvement [29] provides a directed alternative, ensuring a reduction in evaluation steps. This notion is crucial for program optimization, where transformations must be both semantically valid and efficiency-preserving. Despite its significance, proving contextual improvement remains a complex and manual process, relying on pen-and-paper proofs that are tedious and error-prone. Recently, we proposed a theory to address this problem based on a novel rewriting theory of *term evaluation and refinement systems* (TERS), which enables systematic reasoning about evaluation and refinement rules [25]. Within the theory of TERS, we have given sufficient conditions to lead contextual improvement. It involves checking various conditions. The most complex part is *critical pair analysis*, which requires enumerating and checking conflicts between rewrite rules, originally developed in rewriting theory for establishing the confluence property [19, 1]. Although this checking is decidable, it is tedious and error-prone when done manually. The complexity increases significantly as the number of rules grows. In this paper, we introduce a new tool ReCheck, a Haskell-based verification tool that automates the contextual improvement verification.

Organization. The rest of the paper is structured as follows. Section 2 provides an overview of our approach, illustrating how contextual improvement can be formulated and verified using rewriting techniques, and how the tool ReCheck automates it. Section 3 describes the design and implementation of the ReCheck tool. Section 4 presents experimental results on foundational calculi and functional-program benchmarks. Section 5 compares ReCheck with existing verification tools and discusses its limitations and future work.

2 Proving Contextual Improvements in the TERS framework: Rewriting Theory and ReCheck

We explain the theory [25] and how the tool ReCheck automates that sample refinement rules are contextual improvements.

2.1 Rewriting theory

We consider the call-by-value λ-calculus extended with the Maybe monad [36, 22] as used in Haskell. The Maybe monad handles computations that may fail by explicitly representing values as `Just a` (success) or `Nothing` (failure). Using the bind ($\gg=$), operations chain safely, skipping on `Nothing`. It provides pure, side-effect-free failure handling with explicit failure representation. A monad Maybe is a structure with two operations $\mathsf{return} : a \to \mathsf{Maybe}(a)$, $\gg= : \mathsf{Maybe}(a) \to (a \to \mathsf{Maybe}(b)) \to \mathsf{Maybe}(b)$ satisfying the three monad laws. We define a typed λ-calculus extended with the Maybe monad by the ***evaluation rules*** $\mathcal{E}$

$$\lambda(x.M[x])@V \Rightarrow M[V] \qquad (\beta)$$

$$\mathtt{Nothing} \gg= x.F[x] \Rightarrow \mathtt{Nothing} \qquad (\text{bi1})$$

$$\mathtt{Just}(M) \gg= x.F[x] \Rightarrow F[M] \qquad (\text{bi2})$$

$$\mathtt{return}(M) \Rightarrow \mathtt{Just}(M) \qquad (\text{ret})$$

where we use the symbols λ for λ-abstraction and @ for applications. To clarify the distinction of free and bound variables, we write the free variables in capitals, and maintain bound variables as small letters. We use the square brackets $M[x]$ to denote an application of a term to a free variable M. The notation $M[V]$ is ordinarily written as $M[x := V]$, meaning that the variable x appearing in M is replaced with V. In fact, this is the notation of second-order algebraic theory [6, 10, 13]). The (β) is the call-by-value β-reduction rule and the rest are the ordinary definition of the Maybe monad. To make the rewrite system an operational semantics, we define Felleisen-style evaluation contexts E [3]:

$$E ::= \square \mid \mathsf{return}(E) \mid \mathsf{Just}(E) \mid E@t \mid V@E \mid E \gg= t$$

with arbitrary terms t and values $V ::= \lambda(x.t) \mid \mathtt{Just}(V) \mid \mathtt{Nothing}$. The evaluation relation $\to_{\mathcal{E}}$ is generated by the set $\mathcal{E}$ of evaluation rules closed under the evaluation contexts. We consider the monad laws as rewrite rules called ***refinement rules*** $\mathcal{R}$.

$$\texttt{return}(N) \gg\!= y.\, M[y] \Rightarrow M[N] \tag{idL}$$

$$M \gg\!= y.\, \texttt{return}(y) \Rightarrow M \tag{idR}$$

$$(M \gg\!= x.\, K[x]) \gg\!= y.\, L[y] \Rightarrow M \gg\!= x.\, (K[x] \gg\!= y.\, L[y]) \tag{assoc}$$

A scenario in which this could be used as a refinement would be as follows. If a term in a program matches with the left-hand side of one of the above rules, then the compiler rewrites it to the corresponding RHS at compile time. This should be theoretically allowed by the fact that the Maybe monad satisfies the monad laws.

Rewriting theory for deriving contextual improvement. We show that the monad laws can be safely used for optimizing a program with the Maybe monad following the rewriting theoretic method developed in [25]. The pair $(\mathcal{E}, \mathcal{R})$ constitute a ***term evaluation and refinement system (TERS)***. From it, we obtain evaluation $\rightarrow_{\mathcal{E}}$ and refinement $\Rightarrow_{\mathcal{R}}$ on well-typed meta-terms defined by

$$\frac{(l \rightarrow r) \in \mathcal{E} \quad E \in \mathit{Ectx}}{E[l\theta] \rightarrow_{\mathcal{E}} E[r\theta]} \qquad\qquad \frac{(l \Rightarrow r) \in \mathcal{R} \quad C \in \mathit{Ctx}}{C[l\theta] \Rightarrow_{\mathcal{R}} C[r\theta]}$$

where θ is a substitution, Ectx is the set of all evaluation contexts and Ctx is the set of all ordinary contexts. This means that evaluation is only closed under evaluation contexts to make it operational semantics, whereas the refinement is closed under arbitrary context to optimize arbitrary program fragments.

The central problem in this paper is contextual improvement. A set $\mathcal{R}$ of refinement rules is a ***contextual improvement*** w.r.t. a set $\mathcal{E}$ of evaluation rules if the following condition is satisfied. Suppose a refinement rewrite step $s \Rightarrow_{\mathcal{R}} t$. Then for any context C such that $C[s]$ and $C[t]$ are closed terms, $C[s] \overset{k}{\rightarrow}_{\mathcal{E}} v$ implies $C[t] \overset{m}{\rightarrow}_{\mathcal{E}} v'$ such that $v =_{\mathit{Val}} v'$, where v is a value and k and m are the number of evaluation steps such that $k \geq m$ and $=_{\mathit{Val}}$ is an equality on values. This means that the refinement preserves the value v and actually improves the efficiency by reducing the number of evaluation steps. Since refinement is closed under contexts, it is depicted as the following diagram. In general, it is difficult to prove this, because there is universal quantification over all contexts, and it would also be difficult to deal with the evaluation steps and guarantee its decreasing. We have proved the following key theorem.

$$\begin{tikzcd}
C[s] \arrow[rrr, Rightarrow, "\mathcal{R}"'] \arrow[dr, "\mathcal{E}", "k"'] & & & C[t] \arrow[dl, "\mathcal{E}"', "m"] \\
& v \arrow[r, equal, "\mathit{Val}"'] & v' &
\end{tikzcd}$$

Theorem 21 ([25, Thm. 25]) *Let $(\mathcal{E}, \mathcal{R})$ be a TERS. If (1) $\mathcal{E}$ is deterministic, (2) $\mathcal{R}$ is value-invariant and (3) $(\mathcal{E}, \mathcal{R})$ is locally coherent, then refinement $\mathcal{R}$ is a* ***contextual improvement*** *w.r.t. evaluation $\mathcal{E}$.*

This requires to check conditions (1)-(3). We check these for the Maybe monad. The condition (1) of determinism means that the evaluation relation $\to_{\mathcal{E}}$ is deterministic (i.e. if $t_1 \leftarrow_{\mathcal{E}} s \to_{\mathcal{E}} t_2$ then $t_1 = t_2$), which is ensured by the evaluation contexts. The condition (2) of value-invariant means that a refinement of a value is again a value. This is always satisfied in ordinary functional programming languages because values are usually not rewritten by the refinement rules. The condition (3) of ***local coherence*** [25] is the most important and involved to check. It is depicted as the situation (♦). It means the following: suppose a given term s is rewritten in two ways — by evaluating to $E[r\theta]$ (using a rule $l \to r \in \mathcal{E}$ with substitution θ, within an evaluation context E) and by using refinement $\mathcal{R}$ to t, respectively. Then, through further evaluation, they reach w and w' (which are not necessary values) in k- and m-steps, respectively. Moreover, the term w must be refined to w' by the many-step refinement rewriting $\Rightarrow^*_{\mathcal{R}}$ (including zero steps). Local coherence is a condition that for any term s, this situation is satisfied and the evaluation steps are decreasing as (the original steps =) $1 + k \geq m$ (= the evaluation steps after refinement).

(♦) Local coherence

$$\begin{tikzcd}
 & s \arrow[d, phantom, "||"] & \\
 & E[l\theta] \arrow[dl, "\mathcal{E}", "\text{Evaluation}"'] \arrow[dr, Rightarrow, "\mathcal{R}"', "\text{Refinement}"] & \\
E[r\theta] \arrow[d, dotted, "\mathcal{E}"', "k"] & \geq & t \arrow[d, dotted, "m"', "\mathcal{E}"] \\
w \arrow[rr, dotted, Rightarrow, "*", "\mathcal{R}"'] & & w'
\end{tikzcd}$$

Checking all these conditions for all terms in every situation appears to be difficult. Essentially, this represents a conflict between evaluation and refinement. Fortunately, a method of analysis of similar conflicts can be found in rewriting theory. It is called *critical pair analysis* used in the proof method of deriving confluence of rewrite systems [1]. Critical pairs have been used for a different purpose — namely, to show *local confluence*, which is a restricted version of the confluence property. It is a method to enumerate all the conflict rewrites between two overlapped rules and checking their joinability. The "critical pair lemma" concludes that it derives local confluence.

Local coherence can be deduced by calculating *critical pairs*—regarding the two rules as a conflict between evaluation and refinement—and systematically verifying all such cases. This is suitable for automation. This relies on the "critical pair lemma".

Lemma 22 **([25, Thm. 13])** *A well-behaved TERS $(\mathcal{E}, \mathcal{R})$ is locally coherent if and only if all its critical pairs are joinable.*

Definition 23 [25, Def. 12] A TERS $(\mathcal{E}, \mathcal{R})$ is called *well-behaved* if it satisfies:

(i) For any $E \in \mathit{Ectx}$ and $C \in \mathit{Ctx}$, if $E \Rightarrow_{\mathcal{R}} C$ then $C \in \mathit{Ectx}$.

(ii) Every $(l \to r) \in \mathcal{R}$ satisfies: both l and r are linear w.r.t. non-value metavariables. For every non-value metavariable M, if $l[M \mapsto \overline{x}.\Box] \in \mathit{Ectx}$ then $r[M \mapsto \overline{x}.\Box] \in \mathit{Ectx}$.

(iii) For any $(l \to r) \in \mathcal{E}$, l is linear w.r.t. non-value metavariables.

$$\begin{tikzcd}
 & 1 : \underline{\mathsf{Nothing} \gg\!= \mathsf{x.\,return(x)}} \arrow[dl, "(\mathrm{bi1})"'] \arrow[dr, Rightarrow, "(\mathrm{idR})"] & \\
\mathsf{Nothing} \arrow[rr, equal] & & \mathsf{Nothing}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & 2 : \underline{(\mathsf{Nothing} \gg\!= \mathsf{x.\,K[x]}) \gg\!= \mathsf{y.\,L[y]}} \arrow[dl, "(\mathrm{bi1})"'] \arrow[dr, Rightarrow, "(\mathrm{assoc})"] & \\
\mathsf{Nothing} \gg\!= \mathsf{y.\,L[y]} \arrow[dr, "1"] & & \mathsf{Nothing} \gg\!= \mathsf{x.\,(K[x]} \gg\!= \mathsf{y.\,L[y])} \arrow[dl, "1"'] \\
 & \mathsf{Nothing} &
\end{tikzcd}$$

Fig. 1. Critical pairs of the Maybe monad (excerpt)

Checking critical pairs for local coherence. Let us now return to the analysis of the Maybe monad, and see how critical pairs are useful by using Lemma 22. A critical pair is a situation that admits two ways of rewriting by evaluation and refinement, respectively. Careful inspection of $\mathcal{E}$ and $\mathcal{R}$ reveals that it has, in all, 5 patterns of critical pairs. In Fig. 1, we excerpt two patterns, each of which admits the evaluation $\to$ and the refinement $\Rightarrow$.

The pair of divergent terms is called a *critical pair*. The above diagrams show that all the critical pairs are joinable by further evaluation. For example, the critical pair 2: shows an overlap between the rules `(bi1)` and `(assoc)`. Both rules' left-hand sides become the same top term `(Nothing >>= x.K(x)) >>= y.L[y]`, which is obtained by applying the unifier $M \mapsto$ `Nothing`, $\quad F \mapsto$ `x.K(x)` to the left-hand sides. The top term is evaluated (by $\to$) to `Nothing >>= y.L[y]` and refined (by $\Rightarrow$) to `Nothing >>=x.(K[x] >>=y.L[y])`. Both terms are evaluated to `Nothing` by 1 (left term) and 1 (right term) further evaluation steps, respectively. As the conditions of local coherence, the condition (the original evaluated steps $=$) $1 + k \geq m$ ($=$ the evaluation steps after refinement) must be satisfied. In this case, it holds as $1 + 1 \geq 1$, hence the critical pair is joinable. Other cases are similar. As this example shows, critical pairs are calculated by computing the most general unifiers between the left-hand sides of rules. To apply Lemma 22, we need also to check the condition of "well-behavedness" consisting of various syntactic conditions of rule sets. The Maybe monad could be shown to be well-behaved. We conclude that every rule in $\mathcal{R}$ is a contextual improvement.

2.2 Verifying contextual improvement by ReCheck

As the Maybe monad example shows, the most complex part is to exhaustively enumerate all critical pairs and check their joinability. One must try to do the following:

(i) enumerate all the overlaps of the rules without oversight
(ii) rewrite terms to check joinability with counting the number of evaluation steps
(iii) checking various conditions of well-behavedness of rules

These are often quite confusing and prone to error because of similar rules, terms, and variable names. Moreover, one often needs trial and error to develop better refinement rules. It often happens that the rule given is not actually an improvement: possible reasons for not joining are

(i) the rule is wrong (e.g. not preserving values),
(ii) more refinement rules are required to make them contextual improvement, or
(iii) the critical pairs are joinable but the number of evaluation steps is not decreasing (i.e., it is not an optimisation).

In order to devise better refinement rules in such cases, a tool is needed to develop and test new rules quickly. Therefore, we develop a tool ReCheck that automatically checks all the conditions to derive local coherence.

A usage scenario for ReCheck. We show how proofs can be completed by the tool ReCheck. Our system ReCheck is implemented as an embedded domain specific language (DSL) using Template Haskell. The user creates a Haskell script for a given problem by specifying a signature and rules. Then in the Glasgow Haskell interpreter (GHCi), the user attempts several commands to analyse local coherence. The command is realised as a Haskell function. The GHCi interpreter provides an interactive user interface for ReCheck.

Specifying Maybe monad in ReCheck. First, we define the type signature for the Maybe monad:

```
maybeSig = [signature|
  return : a -> Maybe(a); bind : Maybe(a),(a -> Maybe(b)) -> Maybe(b)
  nothing : Maybe(a);     just : a -> Maybe(a) |]
```

Then we define the evaluation rules corresponding to $\mathcal{E}$ by

```
maybeP = [rules|
 (beta) lam(y.M[y]) @ V    => M[V]; (bi1) nothing >>= x.F[x] => nothing
 (bi2)  just(M) >>= x.F[x] => F[M]; (ret) return(M)          => just(M)
 |]
```

and the refinement rules corresponding to $\mathcal{R}$.

```
maybeR = [rules|
 (idL)   return(N) >>= y.M[y] => M[N];   (idR) M >>= y.return(y) => M
 (assoc) (M >>= x.K[x]) >>= y.L[y] => M >>= x.(K[x] >>= y.L[y]) |]
```

We use these as schemas of computation. Therefore, free variables are important as the targets of instantiation by pattern matching. The keyword-headed bracket `[signature|..|]` or `[rules|..|]` indicates the beginning and end of ReCheck's DSL using a feature of Template Haskell.

Values and evaluation contexts by order sorts. We need to define values and evaluation contexts for formulation of TERS. The evaluation rule is not a simple rewrite rule; rather, it is applied within an evaluation context. Moreover, when formalizing values and expressing call-by-value semantics, it becomes necessary to distinguish between "value variables" (which can only be replaced with values) and general variables (which are replaced with arbitrary terms).

This distinction defines *syntax classes* distinct from types. In [25], we formalized TERS as a formal system whose components include such syntactic classes.

In this paper, to implement ReCheck, we refine the original formalization of TERS. Specifically, we employ the notion of *order-sorts* for syntactic classes. The concept of order-sorts was originally introduced in the specification language OBJ [9] and later used to express inheritance relations in object-oriented systems. Syntactic classes such as values must be represented as sorts distinct from types. To this end, we introduce a sort `v` to represent values, corresponding to the syntactic class "values." There is also a sort t for all terms, and since values form a subset of terms, we assume the subsort relation $\mathtt{v} \trianglelefteq \mathtt{t}$. This is precisely why we employ order-sorts.

Rewriting in TERS is formalized so that substitution is applied in a manner that respects this order-sort structure. Furthermore, order-sorts are useful in formalizing evaluation contexts. An evaluation context is typically defined in BNF as a subclass of all contexts. By defining it as a term with the sort `e` (standing for "evaluation context"), we can naturally formulate the terms of evaluation uniformly. Moreover, the check required during rewriting – namely, whether rewriting occurs within an evaluation context – can be implemented as a sort check. Practically, this sort checking can reuse the existing type checker, thereby contributing to an efficient implementation.

Following this methodology, we illustrate the case of the Maybe monad. We have defined values as $V ::= \lambda(x.t) \mid \mathtt{Just}(V) \mid \mathtt{Nothing}$. In ReCheck, this is declared by specifying function symbols having the target sort `v`.

```
maybeValues = [signature| abs : t -> v; nothing : v; just : v -> v |]
```

Likewise, according to the BNF definition of E, we declare function symbols for evaluation contexts,

```
maybeEctx = [signature| xhole : e; return : e -> e; just : e -> e ...|]
```

thereby giving an appropriate definition of terms that constitute evaluation contexts.

Checking local coherence. The tool ReCheck provides automatic checking of local coherence. Invoking the command `improve`, ReCheck first checks the necessary conditions of *well-behavedness* on the rules. Then, ReCheck enumerates all critical pairs as shown in Fig.1, and checks its joinability while rigorously counting the number of evaluations from both sides of each critical pair. It identifies overlaps by applying *higher-order unification* [17, 21], which allows subterms in the left-hand side of one rule to be matched with the root of another. Each critical pair thus obtained corresponds to a divergent peak in the diagrams in Fig.1, where the source term is unified and instantiated accordingly. Using this information, ReCheck then performs a joinability test by exhaustively rewriting both branches and comparing the results. The number of steps taken along each branch is explicitly tracked, and the outcome is either syntactic equality or equivalence modulo refinement steps. This mechanized check ensures that all critical pairs are locally joinable in the precise sense required by contextual improvement. Therefore, by Theorem 21, we conclude that the rule set $\mathcal{R}$ constitutes a contextual improvement over $\mathcal{E}$.

3 Tool ReCheck

Our tool ReCheck is written in Haskell consisting of about 15000 lines of code. It consists of various theory-based components. We discuss each component in turn.

Second-order abstract syntax and DSL ReCheck's specification language based on second-order abstract syntax [7] with metavariable [10, 4] and polymorphism [11, 5]. Second-order abstract syntax cleanly provides the general syntax to describe operational semantics of higher-order languages. It is provided as a domain specific language (DSL), implemented in Template Haskell [31], for writing and modifying TERS specifications interactively. When an automatically checked refinement fails, the user can refine or extend the specification – for example, by adding auxiliary refinement rules analogous to completion steps in rewriting – and immediately re-run verification. This design makes ReCheck not only a verifier but also an exploratory environment for semantic reasoning.

Second-order rewriting engine for evaluation and refinement Evaluation and refinement of TERS are based on polymorphic second-order rewriting [13, 14]. Therefore, we provide second-order rewriting engine by extending the rewriting engine of the previous tool of second-order laboratory SOL [13, 14]. The necessary extension is to respect evaluation contexts. This is implemented by introducing order sorts described in §2.2.

Higher-order pattern unification and matching Second-order rewriting engine requires higher-order pattern matching, which is decidable by imposing Miller's higher-order patterns for the left-hand sides [21]. Moreover, to compute critical pairs, higher-order pattern unification is necessary. this is implemented by following the functional implementation of higher-order pattern unification [12] in Haskell. Pattern matching is then obtained as a special case of unification.

Critical pair enumeration and joinability tests There are two points to note that differ from the usual critical pair analysis in rewriting theory: first, critical pairs are not always possible to generate from overlaps. This requires checking an evaluation context. by sort checking. Another point is the joinability test. The number of evaluations is counted and it is checked that it decreases. Importantly, we do not assume termination of evaluation, so the idea of implementing the joinability test as a comparison of normal forms cannot be used. In the implementation of ReCheck, we limit the joinability test to a certain number of evaluations and terminate it after that. This is reasonable because it is expected that the result (or a term appears on the way to the result) of the refinement will be the same as the original, and it is unlikely that the number of evaluations will be extremely long if the results are the same. This does not affect the soundness. Since the evaluation is deterministic, we only need to look at one reduction path for each evaluation going down for joinability.

Well-behavedness and value-invariance The conditions of well-behavedness of TERS (Def. 23) are checked by directly checking syntactic conditions, such as linearity of variables. Importantly, sort checking is also effectively used to implement the conditions of preservation of evaluation contexts in refinement rules.

Determinism There are two sources of non-deterministic rewriting. (1) when there is more than one rule to match, and (2) the choice of multiple redexes in a term. For (1), we check a condition — known as *orthogonality* [32, Sec.4] in rewriting theory — that there is no overlap between the evaluation rules and that they are left linear. For (2), it is sufficient to check that the evaluation contexts are appropriately defined.

4 Case Studies and Benchmarks

To evaluate the effectiveness of ReCheck, we conducted an extensive benchmark study. Firstly, we consider three examples of foundational calculi for functional programming languages, and demonstrate two aspects of ReCheck: (1) the flexibility of the formalism that faithfully represents different calculi involving own syntax and data: values, monadic computation, let-binding, and λ-bindings, and (2) the power of automation. Secondary, we consider functional program examples and try to compare existing functional program verifiers.

4.1 Simplified computational λ-calculus λ_{ml*}

We consider Sabry and Wadler's computational λ-calculus λ_{ml*} [28, 2], an extension of the simply-typed λ-calculus with `let` and `return` for monadic computation. It forms a TERS [25, Ex.18]. We formalise it in ReCheck and show automatic verification. It has the values, computations and evaluation contexts:

$$\text{Values } V, V' ::= x \mid \lambda(x.P) \quad \text{Evaluation contexts } E ::= \Box \mid \texttt{let}(E, x.P)$$
$$\text{Computations } P, P' ::= \texttt{return}(V) \mid \texttt{let}(P, x.P') \mid V@V'$$

We use the three sorts: values `v`, computations `p`, and terms `t`. We declare the sort declaration `clamSort` for the syntax reflecting the above BNF:

```
clamSort = [signature| lam : t -> v;   xhole : e; let : e,p -> e
 return : v -> p; let : p,p -> p; app : v,v -> p ..|]
```

Note that the original let-syntax `let x = s in t` is represented as a meta-term `let(s,x.t)`, where `x.-` is a variable binding, which is the notation of the second-order abstract syntax [7, 10, 4, 11]. We also declare variables used in the rules to have appropriate sorts, i.e., V for values of sort `v` and P for computations of sort `p`.

```
clamVars = [variables| V : v; P : p |]
```

The λ_{ml*}-calculus consists of 5 rules. We divide it into the evaluation rules `clamP` and the refinement rules `clamR`

```
clamP = [rules|
 (beta)  lam(x.P[x])@V => P[V]; (letBeta) let(return(V),x.P[x]) => P[V]
 |]
clamR = [rules|
 (eta)   lam(x.V@x) => V;       (letEta) let(P, x.return(x)) => P
 (assoc) let(let(P1,x.P2[x]),y.P3[y]) => let(P1,x.let(P2[x],y.P3[y])) |]
```

Now, we consider whether `clamR` is a contextual improvement. From their forms, these rules can be regarded not as computations but rather as optimizations or reorganizations. We invoke the command

```
ReCheck> improve clamSort clamVars clamP clamR
```

in GHCi, which means to check contextual improvement of `clamR` w.r.t. `clamP` under the sort signature `clamSort` with sorted variables `clamVars`. ReCheck firstly checks the necessary syntactic conditions on TERS (determinism, value-invariance, and well-behavedness). Then it automatically enumerates 3 critical pairs, such as the critical pair caused by the overlap between `assoc` and `letBeta`:

$$\begin{tikzcd}
& \texttt{let}(\texttt{let}(\texttt{return}(V), x.P[x]), x'.P'[x']) \arrow[dl, "\text{letBeta}"'] \arrow[dr, Rrightarrow, "\text{assoc}"] & \\
\texttt{let}(P[V], x'.P'[x']) \arrow[dr, equal] & & \texttt{let}(\texttt{return}(V), x.\texttt{let}(P[x], x'.P'[x'])) \arrow[dl, "1"] \\
& \texttt{let}(P[V], x'.P'[x']) &
\end{tikzcd}$$

ReCheck confirms that all the critical pair are joinable with counting the numbers of evaluation steps and checking their decreasing. We conclude that this TERS a contextual improvement. This fact has not been known in the literature [28, 2].

4.2 Effect handlers

We consider a larger example: a calculus for effect handlers [26] as a TERS [25, Ex. 19]. We consider two operations, $\texttt{op}_1$ and $\texttt{op}_0$, and handlers: $\texttt{handler}_1$, which catches the first operation $\texttt{op}_1$, and $\texttt{handler}_0$, which catches no operation. This TERS has more syntactic classes than the λ_{ml*}:

$$\begin{aligned}
\text{Functions } F &::= x \mid \texttt{fun}(x.P) \qquad \text{Values } V ::= \texttt{true} \mid \texttt{false} \mid F \mid H \\
\text{Eval. contexts } E &::= \Box \mid \texttt{do}(E, x.P) \mid \texttt{with}(H, E) \mid \texttt{if}(E, t, t) \mid E@M \mid V@E \\
\text{Handlers } H &::= \texttt{handler}_1(x.P, x.k.P_1) \mid \texttt{handler} \\
\text{Computations } P &::= \texttt{return}(V) \mid \texttt{op}_1(V, y.P) \mid \texttt{op}_0(V, y.P) \mid \texttt{do}(P_1, x.P_2)
\end{aligned}$$

We use the sort `f` for functions, sort `v` for values, and sort `e` for evaluation contexts and declare the sort declaration `clamSort` as

```
handlerSort = [signature|
  fun : p -> f;  fun : p -> v;  handler1 : p,p -> h;  handler0 : p -> h
  op1 : v,p -> p     ;  op0 : v,p -> p;  with : h,p -> p; ... |]
```

We divide the original rules [26] into `handlerP` and `handlerR` (where $i = 0, 1$):

```
handlerP = [rules|
 (1) do(return(V),x.P[x]) => P[V]; (5) fun(x.P[x])@V => P[V]
 (2i) do(opi(V, y.P1[y]), x.P2[x]) => opi(V, y.do(P1[y],x.P2[x]))
 (3) if(true, P1, P2) => P1; (4) if(false, P1, P2) => P2
 (6) with(handler1(x.P[x],x.k.P1[x,k]), return(V)) => P[V]
 (7) with(handler1(x.P[x],x.k.P1[x,k]),op1(V,y.P'[y])) => P1[V,fun(y.P'[y])]
 (8) with(handler1(x.P[x],x.k.P1[x,k]),op0(V,y.P'[y]))
     => op0(V,y.with(handler1(x.P[x],x.k.P1[x,k]),P'[y]))
 (9) with(handler0(x.P[x]),return(V)) => P[V]
 (10i) with(handler0(x.P[x]),opi(V,y.P'[y])) => opi(V,y.with(handler0(x.P[x]),P'[y]))  |]

handlerR = [rules|
 (r3) do(P,x.return(x)) => P;  (r9) fun(x.F@x) => F
 (r4) do(do(P1,x.P2[x]),x.P3[x]) => do(P1, x1.do(P2[x1],x2.P3[x2]))
 (r13) with(handler0(x.P[x]),P') => do(P',x.P[x])   |]
```

The evaluation rules `handlerP` define the operational semantics of effect handlers. In contrast, the rules `handlerR` are refinement rules: they express program transformations that simplify or reorganize handler expressions while preserving their meaning. For example, rule (r3) eliminates a redundant do-expression, and (r13) replaces a handler call by an equivalent sequencing form.

We verify that `handlerR` is a contextual improvement. As the number of rules suggests, critical pair analysis in this case requires many overlap checking. For each rule in `handlerR`, overlaps with the ten `handlerP` rules are computed by higher-order pattern unification with *sort checking* because when overlaps occur, the critical pairs and their joinability *with evaluation-step counting* must be verified. Invoking the `improve` command, ReCheck automatically enumerates *10 critical pairs* and confirms their joinability, such as

$$\begin{tikzcd}
& \mathtt{do}(\mathtt{do}(\mathtt{op}_i(V, x.P[x]), y.P_2[y]), z.P_3[z]) \arrow[dl, "2i"'] \arrow[dr, Rightarrow, "r4"] & \\
\mathtt{do}(\mathtt{op}_i(V, x.\mathtt{do}(P[x], y.P_2[y])), z.P_3[z]) \arrow[d] & & \mathtt{do}(\mathtt{op}_i(V, x.P[x]), y.\mathtt{do}(P_2[y], z.P_3[z])) \arrow[d] \\
\mathtt{op}_i(V, x.\mathtt{do}(\mathtt{do}(P[x], y.P_2[y]), z.P_3[z])) \arrow[rr, Rightarrow, "r4"'] & & \mathtt{op}_i(V, x.\mathtt{do}(P[x], y.\mathtt{do}(P_2[y], z.P_3[z])))
\end{tikzcd}$$

It is a contextual improvement. This fact has not been shown in the literature [26]. Moreover, this also underscores the necessity of automated verification by ReCheck. Given the large number of critical pairs that might otherwise be overlooked in manual analysis.

4.3 Call-by-need λ-calculus

We consider a call-by-need λ-calculus by Maraist, Odersky, Turner, and Wadler [20], reformulated as a TERS [24]. It has 4 syntactic classes assigning sorts `a`,`v`,`r`,`e`:

$$\begin{array}{rll} \text{Answers} & A ::= x \mid \lambda(x.M) \quad \text{Values} & V ::= \lambda(x.M) \mid \texttt{let}(M,x.V) \\ \text{Residuals} & R ::= R@t \mid \texttt{let}(t,r.R) & \\ \text{Evaluation context} & E ::= \Box \mid E@M \mid \texttt{let}(M,x.E) \mid \texttt{let}(E,x.R) & \end{array}$$

The call-by-need λ-calculus consists of four evaluation rules:

```
needP = [rules|
 (I) lam(x.M[x])@N => M[N]; (C) let(A,x.R[x]) => R[A]
 (V) let(M,x.V[x])@N => let(M,x.V[x]@N)
 (A) let(let(M,x.V[x]),y.R[y]) => let(M,x.let(V[x],y.R[y]))  |]
```

This example demonstrates the evaluation generality of ReCheck. It can handle a computation system based on call-by-need, which is neither call-by-value nor call-by-name. The formalization of call-by-need was clarified in works such as [20], where the evaluation is properly controlled by means of let-expressions. ReCheck is capable of formalizing and verifying such a complex evaluation strategy. We take the refinement rules `needR` to be the same `needP` extended with `(eta)` rule

```
needR = needP ++ [rules| (eta) let(M,x.N) => N |]
```

We verify that `needR` is a contextual improvement. Invoking ReCheck, it checks the necessary conditions of TERS and then enumerates two critical pairs and confirms their joinability. Therefore, `needR` is a contextual improvement.

4.4 Lazy program involving infinite lists

We consider a lazy program

```
takeP = [rules|
 (rep1) replicate(z)    => []               ;(take1) take(z, XS)   => []
 (rep2) replicate(s(N)) => s(z):replicate(N);(take2) take(s(N),[]) => []
 (ones) ones => s(z) : ones ; (take3) take(s(N), X:XS) => X : take(N,XS)
```

This is considered as a Haskell program involving infinite lists. It defines basic list operations using Peano natural numbers. `replicate` generates a list of repeated `s(z)`'s, `take` extracts the first N elements from a list, and `ones` denotes an infinite list of ones. We consider a TERS having `takeP` as evaluation rules and the following refinement rule `(conj)`

```
(conj)  take(N, ones) => replicate(N)
```

This means: since the left-hand side of `(conj)` takes N elements from `ones`, it should be valid to rewrite it into N copies of `s(z)`.

This problem was discussed in [18] as a problem of an *inductive theorem* of term rewriting [16, 27]. That is, the question is whether `(conj)` (considered as an equation) can be derived as a theorem if the program is regarded as a set of equational axioms. The problem is called an inductive theorem in the sense that not only equational logic but also induction can be used here. However, all inductive theorem proving methods known so far have required that a given program is terminating or converges to values. Therefore, problems such as (conj) could not be proved by existing methods for rewriting systems. [18] proposed an innovative method for this, which does not assume termination. The problem, however, is that it assumes the condition called local sufficient completeness, which is often involved to derive [18]. In this context, it is interesting to consider whether our method can show this as improvement. ReCheck reports two critical pairs and one non-joinable.

$$\begin{tikzcd}
 & \texttt{take(z,ones)} \arrow[dl, "\text{take}"'] \arrow[dr, Rightarrow, "\text{conj}"] & \\
\texttt{[]} \arrow[dr, equal] & & \texttt{replicate(z)} \arrow[dl, "1"'] \\
 & \texttt{[]} &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \texttt{take(}N\texttt{,ones)} \arrow[dl, "\text{ones}"'] \arrow[dr, Rightarrow, "\text{conj}"] & \\
\texttt{take(}N\texttt{,s(z):ones)} \arrow[rr, Rightarrow, "/" description] & & \texttt{replicate(}N\texttt{)}
\end{tikzcd}$$

This non-joinable pair is a semantically valid rewrite. Therefore, we regard it as a missing rule and incorporate it as an additional refinement rule:

```
(ext) take(N,s(z):ones) => replicate(N)
```

Checking again takeR++ext, ReCheck reports 4 critical pairs and all are joinable. In summary, ReCheck can also verify such lazy problems (with a little effort). This also illustrates another important use of ReCheck. It contributes to "better refinement rule design" by looking at the output even when it fails, rather than just as a verifier. Such "Knuth-Bendix completion [19]"-like method is effective in many cases.

4.5 Functional program benchmarks

As the previous example illustrates, ReCheck can also verify optimization rules for functional programs. This is possible because higher-order functional programs can be represented as TERSs using second-order abstract syntax, which is a simpler structure than fundamental calculi such as the call-by-need λ-calculus. Moreover, any evaluation strategy – including call-by-value, call-by-name, and their combinations – can be specified through evaluation contexts. This allows ReCheck to handle a wider range of programs than ordinary functional programming verifiers. Here we consider the following 9 functional program examples. These benchmarks were chosen to cover a variety of program structures and semantic phenomena – from first-order recursive functions on algebraic datatypes (append (++), `reverse`, `take`, `length`) to higher-order (`map`), lazy (`ones`) and non-terminating (`loop`) programs.

1. `(map/map) map(x.F[x],map(y.G[y],XS)) => map(a.F[G[a]], XS)`
2. `(map/append) map(x.F[x],XS) ++ map(x.F[x],YS) => map(x.F[x], XS ++ YS)`
3. `(assoc) (XS ++ YS) ++ ZS => XS ++ (YS ++ ZS)`
4. `(revrev) reverse (reverse XS) => XS`
5. `(lenPlus) length XS + length YS => length (XS++YS)`
6. `(ones) ns(1) => ones`

7. `(refine) loop => 0`
8. `(copy) copy (copy N) => N`
9. `(make-list) length (make-list M) => M`

Each rule listed above represents a candidate refinement between two higher-order functional program fragments. Whether these rules are indeed contextual improvements is not evident a priori, for instance, the fusion rule `(map/map)` or the associativity rule `(assoc)` could, in principle, alter program behavior depending on the evaluation strategy. The key point is that ReCheck can automatically determine their contextual improvements through critical pair analysis. Several refinement rules require additional explanation.

- `(revrev)` expresses the involutive property of list reversal. This is a typical exercise of equation proved by induction requiring lemmas.
- `(ones)` states a refinement of two infinite lists of 1s, from concrete (`ones`) to abstract (`ns(N)`), where `ones => 1:ones, ns(N) => N:ns(N)`.
- (refine) where `loop` is defined by `loop => loop`. `(refine)` means that non-terminating `loop` is refined to the terminating constant.
- `(copy)` shows that the recursively defined identity function on naturals defined by `copy z = z; copy (s(X)) = s (copy X)` satisfies the idempotency.
- `(make-list) make-list` generates a list of a given length M.

Result. ReCheck successfully verifies that all these refinements as contextual improvements except for `(revrev)`. As in §4.4, once the verification fails, but adding missing rules manually to the set of refinement rules, ReCheck could verify `(revrev)`. This is also reflected in Table 1, which shows the result of trying the same problems `(revrev)` and `(conj)` twice.

4.6 Comparison with other tools for program verification

We are not aware of any other tool that automatically verifies contextual improvement for functional programs and calculi with user-defined operational semantics. As such, ReCheck is unique, making rigorous comparisons with other tools difficult. However, by restricting the scope of problems, ReCheck may be applicable to problems handled by other verification tools, making some comparisons meaningful. We consider three tools. Mochi [30, 34] is a verification tool for OCaml using constrained Horn clauses and automatically checks safety properties using SMT solving. RCaml [33, 35] is a verification tool for OCaml using refinement types that performs SMT-based verification of program invariants. Timbuk [15] is a verification tool based on tree-automata completion. All of them target call-by-value higher-order functional programs.

Setup. For comparison, we applied the examples used in this paper to the three tools MoCHi, RCaml, and Timbuk, when possible, recording whether each tool could verify the property and the verification time. Verification times were measured on an Intel Core i5 13th Gen. CPU with 16GB RAM. All results were obtained using fully automatically without manual proof assistance. For RCaml, two different back-end SMT solvers (PCSat and Spacer) are tested. These tools are not designed for verifying the refinement $s \Rightarrow t$ we have tested so far, therefore, instead we perform verification by treating them as equivalence problems $s \equiv t$. Table 1 summarizes the verification results across benchmarks. Each column lists the verification outcome and the time required for each tool. For ReCheck, we also include the number of critical pairs generated for each problem.

Discussions. The first category (**Foundational calculi**) corresponds to the examples presented in §4.1-4.4. Since these examples go beyond the scope of functional programs, they are outside the range of other tools.

The next category (**Functional programs, our examples**) contains the program examples from §2 and §4. As seen in examples such as `(assoc)` and `(map/map)`, the critical pair analysis in ReCheck can be regarded as an automation of induction over algebraic datatypes. This is similar to the inductionless induction and rewriting induction methods developed in rewriting theory [16, 27]. However, the crucial difference is that our method does not require termination. Proving termination of functional programs or calculi is often extremely difficult. Therefore, it is a major advantage over the rewriting-based induction methods (see also related work discussed in §4.4) for practical verification.

A key difference between ReCheck and the existing tools (MoCHi, RCaml, Timbuk) lies in their treatment of algebraic datatypes (ADTs). While the existing tools perform very well on properties over built-in integers, their SMT encodings make it difficult to reason about ADTs, such as Peano natural numbers or lists. For example, other tools could not verify a fairly standard equality problem on ADTs, such as `(assoc)`,`(map/map)`,`(map/append)`. It is not clear why MoCHi reports unsafe for it. Likewise, it is not clear why `(map/map)` is reported as unsafe or unsat by other tools.

The example `(assocOP)` is the opposite direction of the associativity rule, taken as a refinement rule; in this case, ReCheck correctly reports that the evaluation steps do not decrease. The problem `(loop)` refines the non-terminating function defined by `loop => loop` into the constant `0`. ReCheck can successfully verify such refinements for lazy, non-terminating programs as well.

The last category (**Functional Programs, existing problems**) consists of examples taken from other tools and variants. The problems `(fliter-exist)` and `even` are taken from Timbuk's benchmark. Here, a limitation of ReCheck becomes apparent – its inability to handle integer problems. For instance, `(McCarthy 91)`, the well-known McCarthy's 91 function, which always returns 91, can be verified by SMT-based tools such as MoCHi and RCaml, but ReCheck cannot verify it. Even in the simpler `copy-copy (int)` example, where `copy` copies an integer, ReCheck fails because it does not support built-in integers. However, this also reveals an interesting contrast with SMT-based verifiers. When the same function is rewritten over algebraic datatypes of natural numbers `(copy-copy (nat))`, all other tools except forRCaml (PCSat) fail to verify it, whereas ReCheck succeeds naturally, as expected for such simple algebraic definitions. This experiment illustrates how ReCheck's rewriting-based design provides native support for ADTs, complementing solver-oriented approaches that excel in numeric reasoning.

As for execution time, ReCheck completes all verifications in roughly milliseconds, about two orders of magnitude faster than the other tools. This speed advantage is attributed to the fact that ReCheck does not rely on an SMT solver.

5 Summary and Limitations

Summary. ReCheck provides an automated tool for verifying contextual improvement across diverse evaluation strategies. ReCheck rethinks automated verification by treating operational semantics as user-definable specifications rather than fixed theories. This allows reasoning not only within a language but about the language itself – experimenting with evaluation strategies, algebraic datatypes, and other non-standard features in a unified setting. In this sense, ReCheck serves as a platform for semantic experimentation exploring operational variation.

Current limitations. ReCheck cannot verify improvement established by techniques different from induction, such as foldr/build fusion [8]. Moreover, when verification requires many intermediate lemmas, a single automatic critical pair analysis may not suffice. As shown in §4.4, the verification can then be completed by interactively refining the set of rules through manual additions. The interactive design of ReCheck within GHCi is thus well suited for such iterative refinement of verification rules.

ReCheck currently assumes deterministic and state-less evaluation rules; non-deterministic or probabilistic semantics remain future work. Evaluation strategies and refinement rules must be user-supplied, and the reasoning is purely symbolic without constraint solving, so arithmetic-intensive analyses lie outside its scope. These are deliberate design choices: ReCheck complements SMT solver-based systems by extending automated reasoning into the semantic domain rather than numeric reasoning.

Future directions. Planned extensions include supporting for effectful and non-deterministic semantics, and integration with SMT or proof-assistant backends. We also plan to provide an extensible library of TERS specifications for common functional calculi, establishing ReCheck as a shared infrastructure for designing and analysing rewriting and operational semantics.

Acknowledgments. We are grateful to Koko Muroya, Shotaro Karasaki and Tomoya Ichikawa for valuable discussions and collaboration on the foundational theory and applications of ReCheck. This research was partially supported by a grant from the Mitsubishi Foundation (Hamana), and by JSPS KAKENHI Grant Numbers 24K02919 (Hamana) and 24K14898 (Emoto).

No.	Problem	ReCheck	Critical	MoCHi	RCaML	RCaML	Timbuk
			pairs		(PCSat)	(Spacer)	
Fundational calculi							
1	Maybe monad	YES 0.0051s	5	not supported	not supported	not supported	not supported
2	λ_{ml*}-calculus	YES 0.0016s	3	not supported	not supported	not supported	not supported
3	Effect handlers	YES 0.0096s	10	not supported	not supported	not supported	not supported
4	Call-by-need λ-calculus	YES 0.0021s	2	not supported	not supported	not supported	not supported
Functional progams (our examples)							
5	map/map	YES 0.0014s	2	Unsafe 1.739s	Unsat 0.351s	Unsat 0.195s	NT
6	assoc	YES 0.0010s	2	Unsafe 0.944s	NT	NT	NT
7	assocOp	MAYBE 0.0011s	2	Unsafe 1.261s	NT	NT	NT
8	map/append	YES 0.0018s	2	Exception: Stack overflow	Unsat 1.016s	Unsat 0.288s	NT
9	lenPlus	YES 0.0012s	2	Exception: Stack overflow	NT	SegFault 45.16s	NT
10	conj1	MAYBE 0.0010s	2	not supported	Sat 0.19s	Sat 0.197s	NT
11	conj2	YES 0.0016s	4				
12	ones	YES 0.0004s	1	Safe 0.332s	Sat 0.063s	Sat 0.073s	NT
13	refine	YES 0.0002s	1	Safe 0.386s	Sat 0.037s	Sat 0.046s	ERROR 0.094s
14	revrev1	MAYBE 0.0008s	2	Unsafe 1.242s	NT	OutOfMem 68.036s	NT
15	revrev2	YES 0.0023s	4				
Functional progams (existing benchmark problems and variants)							
16	copy-copy (int)	not supported		Safe 0.446s	Sat 0.823s	NT	
17	copy-copy (nat)	YES 0.0005s	2	not supported	Sat 0.185s	NT	NT
18	makelist (int)	not supported		Safe 0.938s	NT	NT	
19	makelist (nat)	YES 0.0004s	1	not supported	NT	NT	NT
20	filter-exists	MAYBE 0.0034s	2	Unsafe 1.193s	Unsat 0.291s	Unsat 0.193s	Proved 1.163s
21	even	YES 0.0006s	3	failure	Unsat 10.624s	Unsat 0.1s	Proved 0.343s
22	McCarty91	not supported		Safe 0.619s	Sat 0.915s	Sat 0.083s	NT

Yes$\cdots$Contextual improvement, MAYBE$\cdots$Unknown, Sat$\cdots$Satisfiable, Unsat$\cdots$Unsatisfiable, NT$\cdots$Non-Termination, Blank$\cdots$Not tested

Table 1. Experiments

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
2. Faggian, C., Guerrieri, G., Treglia, R.: Evaluation in the computational calculus is non-confluent. In: 10th International Workshop of Confluence, IWC 2021. pp. 31–36 (2021)
3. Felleisen, M., Friedman, D.P., Kohlbecker, E.E., Duba, B.F.: A syntactic theory of sequential control. Theor. Comput. Sci. **52**, 205–237 (1987)
4. Fiore, M.: Second-order and dependently-sorted abstract syntax. In: Proc. of LICS'08. pp. 57–68 (2008)
5. Fiore, M., Hamana, M.: Multiversal polymorphic algebraic theories: Syntax, semantics, translations, and equational logic. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013. pp. 520–529 (2013)
6. Fiore, M., Mahmoud, O.: Second-order algebraic theories. In: Proc. of MFCS'10. pp. 368–380. LNCS 6281 (2010)
7. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: Proc. of LICS'99. pp. 193–202 (1999)
8. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Proceedings of the conference on Functional programming languages and computer architecture. pp. 223–232 (1993)
9. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. **105**(2), 217–273 (1992). https://doi.org/10.1016/0304-3975(92)90302-V, https://doi.org/10.1016/0304-3975(92)90302-V
10. Hamana, M.: Free Σ-monoids: A higher-order syntax with metavariables. In: Proc. of APLAS'04. pp. 348–363. LNCS 3302 (2004)
11. Hamana, M.: Polymorphic abstract syntax via Grothendieck construction. In: FoSSaCS'11. pp. 381–395. LNCS3467 (2011)
12. Hamana, M.: A functional implementation of function-as-constructor higher-order unification. In: Proc. of International Workshop on Unification 2017. pp. 10–15 (2017), co-located with FSCD'17, Oxford, UK
13. Hamana, M.: How to prove decidability of equational theories with second-order computation analyser SOL. Journal of Functional Programming **29**(e20) (2019)
14. Hamana, M., Abe, T., Kikuchi, K.: Polymorphic computation systems: Theory and practice of confluence with call-by-value. Sci. Comput. Program. **187**, 102322 (2020). https://doi.org/10.1016/J.SCICO.2019.102322, https://doi.org/10.1016/j.scico.2019.102322
15. Haudebourg, T., Genet, T., Jensen, T.P.: Regular language type inference with term rewriting. PACMPL **4**(ICFP), 112:1–112:29 (2020)
16. Huet, G., Hullot, J.M.: Proofs by induction in equational theories with constructors. Journal of computer and system sciences **25**(2), 239–266 (1982)
17. Huet, G.P.: A unification algorithm for typed lambda-calculus. Theor. Comput. Sci. **1**(1), 27–57 (1975)
18. Kikuchi, K., Aoto, T., Sasano, I.: Inductive theorem proving in non-terminating rewriting systems and its application to program transformation. In: Proc. of PPDP 2019. pp. 13:1–13:14. ACM (2019)
19. Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: Computational Problem in abstract algebra, pp. 263–297. Pergamon Press, Oxford (1970)

20. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-name, call-by-value, call-by-need and the linear lambda calculus. Theor. Comp. Sci. **228**(1-2), 175–210 (1999). https://doi.org/10.1016/S0304-3975(98)00358-2
21. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of Logic and Computation **1**(4), 497–536 (1991)
22. Moggi, E.: Notions of computation and monads. Information and Computation **93**, 55–92 (1991)
23. Morris Jr, J.H.: Lambda-calculus models of programming languages. Ph.D. thesis, Massachusetts Institute of Technology (1969)
24. Muroya, K., Hamana, M.: Typed term evaluation systems with refinements for contextual improvement, submitted for publication
25. Muroya, K., Hamana, M.: Term evaluation systems with refinements: First-order, second-order, and contextual improvement. In: Proc. of 17th International Symposium on Functional and Logic Programming (FLOPS'24). pp. 31–61. LNCS 14659 (2024)
26. Pretnar, M.: An introduction to algebraic effects and handlers. invited tutorial paper. In: Ghica, D.R. (ed.) The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015. Electronic Notes in Theoretical Computer Science, vol. 319, pp. 19–35. Elsevier (2015). https://doi.org/10.1016/J.ENTCS.2015.12.003, https://doi.org/10.1016/j.entcs.2015.12.003
27. Reddy, U.S.: Term rewriting induction. In: International Conference on Automated Deduction. pp. 162–177. Springer (1990)
28. Sabry, A., Wadler, P.: A reflection on call-by-value. In: Harper, R., Wexelblat, R.L. (eds.) Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996. pp. 13–24. ACM (1996). https://doi.org/10.1145/232627.232631, https://doi.org/10.1145/232627.232631
29. Sands, D.: Total correctness by local improvement in the transformation of functional programs. ACM Trans. Program. Lang. Syst. **18**(2), 175–234 (1996). https://doi.org/10.1145/227699.227716
30. Sato, R., Unno, H., Kobayashi, N.: Mochi: Software model checker for a higher-order functional language (demo presentation). In: 2012 ACM SIGPLAN Workshop on ML (ML 2012) (2012)
31. Sheard, T., Jones, S.P.: Template metaprogramming for Haskell. In: Proc. Haskell Workshop 2002 (2002)
32. Terese: Term Rewriting Systems. No. 55 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (2003)
33. Unno, H., Kobayashi, N.: On-demand refinement of dependent types. In: Garrigue, J., Hermenegildo, M.V. (eds.) Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4989, pp. 81–96. Springer (2008)
34. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 75–86. ACM (2013). https://doi.org/10.1145/2429069.2429081, https://doi.org/10.1145/2429069.2429081

35. Unno, H., Torii, S., Sakamoto, H.: Automating induction for solving horn clauses. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 571–591. Springer (2017)
36. Wadler, P.: Comprehending monads. In: ACM Conference on Lisp and Functional Programming. pp. 61–78. Nice, France (June 1990)