GSOL: A Confluence Checker for Haskell Rewrite Rules

Yao Faustin Date and Makoto Hamana

[1] Department of Computer Science, Gunma University, Japan
t201d047@gunma-u.ac.jp
[2] Faculty of Informatics, Gunma University, Maebashi, Japan
hamana@cs.gunma-u.ac.jp

Abstract. We present a tool GSOL, a confluence checker for GHC. It checks the confluence property for rewrite rules in a Haskell program by using the confluence checker SOL (Second-Order Laboratory). The Glasgow Haskell Compiler (GHC) allows the user to use rewrite rules to optimize Haskell programs in the compilation pipeline. Currently, GHC does not check the confluence of the user-defined rewrite rules. If the rewrite rules are not confluent then the optimization using these rules may produce unexpected results. Therefore, checking the confluence of rewrite rules is important. We implement GSOL using the plugin mechanism of GHC and provide three usages: (1) a stand-alone command `gsol`, (2) checking by Cabal building, and (3) a Web interface http://solweb.mydns.jp/. We demonstrate confluence checking of the rewrite rules in the Arrow library.

1 Introduction

The Glasgow Haskell Compiler (GHC) [MJ12] is an open source compiler and interactive environment for the functional language Haskell [Mar10]. It has builtin transformation rules to optimize Haskell programs during the compilation [JS98]. The user can also add rewrite rules to a program to specify optimizing transformations [JTH01]. The notion of *confluence* is one of the important properties of rewrite rules known in the theory of rewriting [Hue80]. Confluence guarantees the uniqueness of normal forms, which is particularly desirable in functional programming. However, GHC does not attempt to check the confluence of user-defined rewrite rules.

In this paper, we present GSOL, a GHC plugin to check the confluence of rewrite rules in a Haskell program. It uses a confluence checker, Second-Order Laboratory (SOL) [Ham19,HAK20]. To illustrate our work, we consider the following Haskell program that involves two rewrite rules.

```
module F where                          e = f 99

{-# RULES                               {-# NOINLINE f #-}
        "f/0" forall x. f x = 0         f  :: Integer -> Integer
        "f/1" forall x. f x = 1         f x = 0
 #-}
```

The code within the {`-# ... #-`} is called a *pragma*[3]. In the `RULES` pragma, there are two rules named `"f/0"` and `"f/1"`. If the the compiler chooses to apply the rule `"f/0"` then the expression `f 99` is rewritten to 0. If it chooses to apply the rule `"f/1"` then the expression `f 99` is rewritten to 1. Therefore, they are not confluent. But the GHC compiler *does not* notice this non-confluence.

To the best of our knowledge, there is no tool to check the confluence of rewrite rules directly from a Haskell program. In the field of term rewriting, a few confluence checkers for *higher-order* rewrite systems have been developed [OKAT17,NFM17,Ham19]. We use the tool SOL [Ham19,HAK20] to check the confluence of GHC rewrite rules. Since SOL has been shown to be the strongest tool among the existing confluence tools that participated in the Higher-Order Rewriting category of the International Confluence Competition 2018 [AHH$^+$18] and 2020 [CoC20], we believe that this is the best choice for confluence checking of Haskell rewrite rules.

Related work. Rewrite rules have been used as a way to automate the optimization process of functional programs [HL$^+$20,JTH01,PCMK12,SFLD15]. We mention two recent works.

In [SFLD15], Steuwer et al. applied rewrite rules to transform a high-level functional program into a low-level functional representation from which OpenCL code is generated. They showed that this approach offered performance on a par with highly tuned code for multi-core CPUs and GPUs written by experts.

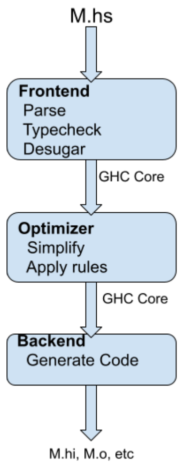
In [HL$^+$20], a high-level program H was rewritten into an equivalent program but with lower lever constructs L. Then the program L went through code generation to produce platform specific program. The language of rewriting strategies ELEVATE was used to rewrite high-level RISE program into low-level RISE program. ELEVATE forced the user to specify the rules to apply and the order of application. Hence they avoided the issue of non-confluence.

This work. In the previous works on rewrite rules for optimizations including [HL$^+$20,JTH01,PM$^+$05,PCMK12,SFLD15], confluence and termination of rewrite rules have not been checked automatically although ensuring them has been recognized as an important problem. In this work, we solve this problem by applying the result of well-established rewriting technology to the real-world functional programming language Haskell. We use an automatic confluence checker SOL to check the confluence of GHC rewrite rules in a Haskell program.

Organisation. This paper is organised as follows. In §2, we first explain necessary background for developing GSOL. We then explain our implementation in §3. In §4, we demonstrate an example of arrows. In §5, we discuss future work.

[3] The `NOINLINE` pragma instructs the compiler not to expand `f 99` by using the function definition. Without this indication, `f 99` is inlined to `0` before applying the rewrite rules, resulting that no rewrite rules are fired.

2 Background



GHC. The compilation process of a Haskell program consists of three big steps: *frontend*, *optimizer*, and *backend*. The frontend consists of parsing, type checking and the transformation into the GHC's intermediate language called *GHC Core*, implementing System F_C [MMJK07]. A GHC Core expression consists of variables, literals, abstractions, applications and variable bindings. The optimizer optimizes the GHC Core program through various transformations. The *simplifier* implements most of those transformations using a set of builtin rules [JS95,JS98,JWS96]. The simplifier can also use rewrite rules specified in the program. The optimization of a GHC Core program is divided in a series of Core-to-Core translations. The simplifier is one of them. The role of the backend is to generate code for different platforms.

GHC plugins. The plugin mechanism of GHC [GHC20] allows programmers to insert their own passes in the compilation pipeline. We use it to implement a confluence checker. Our plugin receives a Core program, checks local confluence and termination, and outputs the result of the checks but does not modify the Core program.

GHC rewrite rules. The user can use rewrite rules in a Haskell program to teach the compiler optimizing transformations specific to their programs [JTH01]. The syntax of rewrite rules is:

```
{-# RULES
   "name" forall <var>...<var>. f <expr> = <expr>
   ...
 #-}
```

The left-hand side of a rule must be a function application `f <expr>` where the function `f` is in the scope. The left-hand side and right-hand side of the rewrite rules are parsed as Core expressions at the compile time and forms rewrite rules on Core, which we call ***Core rules***.

Notion of confluence. *Term rewriting* [BN98,Ter03] is a research field of theoretical computer science. It studies rewrite relations on term structures using various relational, order theoretic, and algebraic methods. There are two important properties of rewrite relation, namely, *termination* and *confluence*. Termination (which means strong normalisation) is to reach the normal form in finite time by any way of rewriting. Confluence (CR) is a property of the rewrite relation, stating that any two divergent computation paths are joinable, as shown in the diagram. Confluence ensures the existence of unique normal forms, which is desirable in functional programming.

$$\begin{tikzcd} & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\ M_1 \arrow[dr, "*"'] & \text{CR} & M_2 \arrow[dl, "*"] \\ & N & \end{tikzcd} \qquad \begin{tikzcd} & M \arrow[dl] \arrow[dr] & \\ M_1 \arrow[dr, "*"'] & \text{LCR} & M_2 \arrow[dl, "*"] \\ & N & \end{tikzcd}$$

To deduce confluence, Newman's lemma is useful [Hue80]. It states "termination and local confluence implies confluence". Local confluence (LCR) is a weakened variant of the confluence property that states that if there is two (different) ways of one step rewriting "$\to$" from a term M, then there exists a term N, to which the divergent terms M_1 and M_2 can be rewritten by many step rewriting "$\to^*$".

To prove local confluence, we should check all possible situations that admit two ways of rewriting, and also to check their convergence. Instead of examining possibly infinite number of such situations, it has been shown that checking the joinability of finite number of critical divergent terms, called *critical pairs*, is enough to conclude local confluence [BN98]. Critical pairs can be enumerated by computing overlaps between the left-hand sides of rules using high-order unification. For example, there is a critical pair $(1, 0)$ in the rewrite rules of the module `F.hs` given in Introduction.

Our tool GSOL can automatically show it as follows:

```
******** Critical pairs ********
1: Overlap (1)-(2)--- X'|-> X ---------------------
   (1) |f(X)| => 1
   (2) f(X') => 0
                                  f(X)
                          1 <-(1)-/\-(2)-> 0
                           ---> 1 =#= 0 <---
#NON 1 joinable... (Total 1 CPs)
..
NO
```

This shows that the left-hand sides `f(X)`, `f(X')` of the rules are unifiable by the unifier $\{$`X'` $\mapsto$ `X`$\}$. It produces a term `f(X)`, which is rewritten to two different terms $1, 0$ forming a critical pair. If these can be rewritten to a common term, then we conclude local confluence. But in this case, these are already different normal forms. Therefore, this non-joinability is an evidence of *non-confluence*, hence this outputs `NO`.

SOL. SOL is an implementation of a formal framework of *second-order computation systems*, which is a computational counterpart of second-order algebraic theories [FH10,FM10]. This framework has been used in [Ham19].

Second-order computation systems are based on second-order abstract syntax given by the language of meta-terms [Ham04]:

$$t ::= x \mid x.t \mid f(t_1, \ldots, t_n) \mid M[t_1, \ldots, t_n].$$

These forms are respectively variables, abstractions, and function terms, and the last form is called a meta-application. A meta-application $M[t_1, \ldots, t_n]$ means: when we instantiate M with a term s, free variables of s are replaced with (meta-)terms $t_1, \ldots, t_n$.

The meta-terms have second-order types [Ham19]. Computation rules are pairs of meta-terms. Appendix A provides a complete definition.

```
{-# RULES
"compose/arr"    forall f g. (arr f) . (arr g)       = arr (f . g)
"first/arr"      forall f.   first (arr f)           = arr (first f)
"compose/first"  forall f g. (first f) . (first g)   = first (f . g)
"product/arr"    forall f g. arr f *** arr g         = arr (f *** g)
...
#-}
```

Fig. 1. Rewrite rules in `Control.Arrow` (excerpt)

3 Implementation

We implemented GSOL as a Core plugin to check the confluence of GHC rewrite rules. Our plugin is installed into the beginning of the optimization pipeline. The plugin proceeds as the following three steps:

1. Collecting the Core rules.
2. Translating them to SOL rules.
3. Calling SOL for checking. SOL performs the checking functions and print the output to the standard output.

The translation of Core rules is done by applying a structural recursive translation of Core terms to both sides of each rule. It is basically a known encoding method used in [Ham19,HAK20], which encodes λ-terms to meta-terms (cf. §B).

We provided three ways to use GSOL: (1) a stand-alone shell command `gsol` for a single Haskell file, (2) checking all files in a Cabal package by specifying options and using the `cabal build` command, and (3) a Web interface.

4 Example: Arrow

In this section, we demonstrate GSOL by examining the Arrow library of GHC. Arrows [Hug00,Pat01] provide a way to programming with various computational effects in Haskell. `Control.Arrow` is a library in the Haskell base package. Arrows are implemented using a type class:

```
class Category a => Arrow a where
   arr    :: (b -> c) -> a b c
   first  :: a b c -> a (b,d) (c,d)
...
```

Instances of the arrow class satisfy various laws. Most important laws are the laws of Freyd category ([Pat01, Fig. 1: Arrow equations]), which come from the semantics of arrows [HJ06], hence are they valid for any instance of arrows. Slight different specific laws have been described in `Control.Arrow` as GHC rewrite rules, which are excerpted in Fig. 1. Note that the file also includes other extensions including `ArrowChoice` and their laws[4]. We try to check the confluence of them.

[4] Notice that the rewrite rules defined there are valid only when the instance is `a = (->)`. But it is not mentioned in the file `Control.Arrow`. We do not know why such specific laws (rather than the Freyd category laws) were described as rewrite rules.

(1) Stand-alone command. We check the joinability of critical pairs in the `Control.Arrow` library in the shell by invoking the command:

```
> gsol cri Control_Arrow.hs
```

It reports 8 critical pairs and all are non-joinable (cf. Fig. 6), such as:

$$\begin{tikzcd} & 7 : \underline{\mathsf{first}(\mathsf{arr}(\mathsf{x.f[x]}))} \,.\, \mathsf{first}(\mathsf{g}) \arrow[dl, "\text{(compose/first)}"'] \arrow[dr, "\text{(first/arr)}"] & \\ \mathsf{first}(\mathsf{arr}(\mathsf{x.f[x]}) \bullet \mathsf{g}) \arrow[r, phantom, "\neq"] & & \mathsf{arr}(\mathsf{first}(\mathsf{y.f[y]})) \bullet \mathsf{first}(\mathsf{g}) \end{tikzcd}$$

Since these are normal forms, it shows *non-confluence* of the rewrite rules in `Control.Arrow`, which has not been reported elsewhere[5].

An intended scenario of the usage of GSOL is that if the user receives this kind of information by applying GSOL, then the user tries to fix it by modifying the rules or adding new rules. In this respect, reporting the non-confluence information is also an important feature of GSOL.

Generally, the command `gsol` has two options: `cri` for critical pair checking to show local confluence, and `sn` for termination checking. If no options are given `gsol` checks both local confluence and termination. If the rules are locally confluent and terminating then they are confluent. The command `gsol` is implemented as invoking GHC with the plugin `SOL.Plugin`.

(2) Cabal. GSOL can also check the confluence of files provided as a Cabal packages. We consider a sample package `myarrows` configuring the `Control.Arrow` library. The file `arrows.cabal` should contain the configuration as:

```
library myarrows
  exposed-modules:  Control.Arrow
  build-depends:    base ^>=4.13.0.0, SOL
  ghc-options:      -fplugin=SOL.Plugin -fplugin-opt=SOL.Plugin:cri
```

which also requires the `SOL` package. Invoking the command `cabal build`, GSOL is automatically called to check local confluence. One can also check termination by `-fplugin-opt=SOL.Plugin:sn`.

(3) Web interface. To ease the checking with GSOL, we have also developed a web interface, which is available at `http://solweb.mydns.jp/` . Several examples are already available. Choosing the file `Control_Arrow.hs` from the pull-down menu, and "GHC rule" format and "WCR" buttons, the user can get the result (cf. the screenshot, Fig. 7).

One can also check the Freyd category laws by choosing the file `MyArrow.hs`.

[5] This can be joinable by adding a new rule `forall f. arr(f) = f`, which is valid again only when the instance is `a = (->)`.

5 Summary and Future Work

In this paper, we presented a tool GSOL, a confluence checker for GHC. It checks the confluence property for rewrite rules in a Haskell program by using SOL. We implemented GSOL using the plugin mechanism of GHC and provided three usages: (1) a stand-alone command, (2) Cabal, and (3) a Web interface. We demonstrated confluence checking of the rewrite rules in the Arrow library.

As a future work, we will improve the type translation of Core rules to SOL to deal with type constraints and type variables more properly. In Core rules, type constraints in type signature in the original program become type parameters. The current translation in GSOL drops the type parameters, which suffices for critical pair checking, but makes termination checking weaker. We need to investigate to solve this issue. The framework of polymorphic rewrite rules [HAK20] would be useful.

We believe that our development is also applicable to the termination and confluence checking of type functions [CKJM05,CKJ05]. We plan to apply the technology developed in this paper to them.

Recently, rewrite rules and checking their confluence have been an important topic in dependently typed programming languages [Bla20,CTW21]. We also plan to apply our technology to this field.

References

AHH+18. T. Aoto, M. Hamana, N. Hirokawa, A. Middeldorp, J. Nagele, N. Nishida, K. Shintani, and H. Zankl. Confluence Competition 2018. In *Proc. of FSCD 2018*, volume 108 of *LIPIcs*, pages 32:1–32:5, 2018.

Bla20. F. Blanqui. Type safety of rewrite rules in dependent types. In *Proc. of FSCD 2020*, volume 167 of *LIPIcs*, pages 13:1–13:14, 2020.

BN98. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

CKJ05. M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *Proc. of ICFP'05*, pages 241–253, 2005.

CKJM05. M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *Proc. of POPL'05*, pages 1–13, 2005.

CoC20. Confluence competition official site, 2020. http://project-coco.uibk.ac.at/2020/.

CTW21. J. Cockx, N. Tabareau, and T. Winterhalter. The taming of the rew: a type theory with computational assumptions. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

FH10. M. Fiore and C.-K. Hur. Second-order equational logic. In *Proc. of CSL'10*, LNCS 6247, pages 320–335, 2010.

FM10. M. Fiore and O. Mahmoud. Second-order algebraic theories. In *Proc. of MFCS'10*, LNCS 6281, pages 368–380, 2010.

GHC20. Compiler plugins, 2020. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/extending_ghc.html.

HAK20. M. Hamana, T. Abe, and K. Kikuchi. Polymorphic computation systems: Theory and practice of confluence with call-by-value. *Science of Computer Programming*, 187(102322), 2020.

Ham04. M. Hamana. Free Σ-monoids: A higher-order syntax with metavariables. In *Proc. of APLAS'04*, LNCS 3302, pages 348–363, 2004.

Ham19. M. Hamana. How to prove decidability of equational theories with second-order computation analyser SOL. *Journal of Functional Programming*, 29(e20), 2019.

HJ06. C. Heunen and B. Jacobs. Arrows, like monads, are monoids. In *Proc. of MFPS 22, ENTCS*, volume 158, pages 219–236, 2006.

HL$^+$20. B. Hagedorn, J. Lenfers, et al. Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.*, 4(ICFP), 2020.

Hue80. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of ACM*, 27(4):797–821, 1980.

Hug00. J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

JS95. S. P. Jones and A. Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, pages 184–204. Springer, 1995.

JS98. S. P. Jones and A. Santos. A transformation-based optimiser for haskell. *Science of computer programming*, 32(1-3):3–47, 1998.

JTH01. S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop 2001*, 2001.

JWS96. S. P. Jones, P. Will, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 1–12, 1996.

Mar10. S. Marlow. Haskell 2010 language report, 2010.

MJ12. S. Marlow and S. P. Jones. *The Glasgow Haskell Compiler*, volume 2. Lulu, January 2012.

MMJK07. S. Martin, C. Manuel, S. P. Jones, and D. Kevin. System f with type equality coercions. In *Proc. of TLDI '07*, pages 53–66, 2007.

NFM17. J. Nagele, B. Felgenhauer, and A. Middeldorp. CSI: New evidence – a progress report. In *Proc. of CADE'17*, LNCS (LNAI) 10395, pages 385–397, 2017.

OKAT17. K. Onozawa, K. Kikuchi, T. Aoto, and Y. Toyama. ACPH: System description. In *6th Confluence Competition (CoCo 2017)*, 2017.

Pat01. R. Paterson. A new notation for arrows. In *Proceedings of ICFP'07*, pages 229–240, 2001.

PCMK12. A. Panyala, D. Chavarria-Miranda, and S. Krishnamoorthy. On the use of term rewriting for performance optimization of legacy HPC applications. In *Proc. International Conference on Parallel Processing*, pages 399–409, September 2012.

PM$^+$05. M. Püschel, J. M. F. Moura, et al. SPIRAL: code generation for DSP transforms. *Proc. IEEE*, 93(2):232–275, 2005.

SFLD15. M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proc. of ICFP'15*, 2015.

Ter03. Terese. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

Appendix

In this appendix, we review second-order computation systems, which is the foundation of SOL, and describe the translation

A Second-Order Computation Systems

We review a formal framework of second-order computation based on second-order algebraic theories [FH10,FM10]. This framework has been used in [Ham19].

Types. We assume that $\mathcal{A}$ is a set of *atomic types* (e.g. Bool, Nat, etc.). We also assume a set of *type constructors* together with arities $n \in \mathbb{N}$, $n \geq 1$. The sets of ***molecular types*** $\mathcal{T}_0$ and ***types*** $\mathcal{T}$ are generated by the following rules:

$$\frac{b \in \mathcal{A}}{b \in \mathcal{T}_0} \qquad \frac{\begin{array}{l} b_1, \ldots, b_n \in \mathcal{T}_0 \\ T\ n\text{-ary type constructor} \end{array}}{T(b_1, \ldots, b_n) \in \mathcal{T}_0} \qquad \frac{a_1, \ldots, a_n, b \in \mathcal{T}_0}{a_1, \ldots, a_n \to b \in \mathcal{T}}$$

Remark A1 Molecular types work as "base types" in ordinary type theories. But in our usage, we need "base types" which are constructed from "more basic" types. Hence we first assume atomic types as the most atomic ones, and then generate molecular types from them.

Terms. A ***signature*** Σ is a set of function symbols of the form

$$f : (\overline{a_1} \to b_1), \ldots, (\overline{a_m} \to b_m) \to c$$

where all a_i, b_i, c are mol types (thus any function symbol is of up to second-order type). A sequence of types may be empty in the above definition. The empty sequence is denoted by $()$, which may be omitted, e.g., $b_1, \ldots, b_m \to c$, or $() \to c$. The latter case is simply denoted by c. We assume two disjoint syntactic classes of letters, called ***metavariables*** (written as capital letters $M, N, K, \ldots$) and ***variables*** (written usually $x, y, \ldots$). The raw syntax is given as follows.

- ***Terms*** have the form $t ::= x^a \mid x.t \mid f(t_1, \ldots, t_n)$.
- ***Meta-terms*** extend terms to $s ::= x^a \mid x.s \mid f(s_1, \ldots, s_n) \mid M[s_1, \ldots, s_n]$.

These forms are respectively *variables*, *abstractions*, and *function terms*, and the last form is called a *meta-application*. We may write $x_1^{a_1}, \ldots, x_n^{a_n}.t$ (or $\overline{x^a}.t$) for $x_1^{a_1}. \cdots .x_n^{a_1}.t$, and we assume ordinary α-equivalence for bound variables. Hereafter, we often omit the superscript of variables $x_i^{a_i}$. We also assume that every bound variable and free variable are mutually disjoint in computation steps to avoid α-renaming during computation. If computation rules do not satisfy this property, we consider suitable variants of the rules by renaming free/bound (meta)variables. A metavariable context Z is a sequence of (metavariable:type)-pairs, and a context Γ is a sequence of (variable:mol type)-pairs. A judgment is of the form

$$Z \rhd \Gamma \vdash t : b.$$

$$\frac{y : b \in \Gamma}{Z \rhd \Gamma \vdash\ y\ : b} \qquad \frac{\begin{array}{c}(M : a_1, \ldots, a_m \to b) \in Z \\ Z \rhd \Gamma \vdash\ t_i\ : a_i \quad (1 \le i \le m)\end{array}}{Z \rhd \Gamma \vdash\ M[t_1, \ldots, t_m]\ : b} \qquad \frac{\begin{array}{c}f : (\overline{a_1} \to b_1), \cdots, (\overline{a_m} \to b_m) \to c \in \Sigma \\ Z \rhd \Gamma, \overline{x_i : a_i} \vdash\ t_i\ : b_i \quad (1 \le i \le m)\end{array}}{Z \rhd \Gamma \vdash\ f(\,\overline{x_1^{a_1}}.t_1, \ldots, \overline{x_m^{a_m}}.t_m\,)\ : c}$$

Fig. 2. Typing rules of meta-terms

$$\text{(Rule)}\frac{\begin{array}{c}\rhd \Gamma', \overline{x_i : a_i} \vdash\ s_i\ : b_i \quad (1 \le i \le k) \quad \theta = [\,\overline{M \mapsto \overline{x}.s}\,] \\ (M_1 : (\overline{a_1} \to b_1), \ldots, M_k : (\overline{a_k} \to b_k) \rhd\ \vdash\ \ell \Rightarrow r\ : c) \in \mathcal{C}\end{array}}{\rhd \Gamma' \vdash\ \theta^\sharp(\ell) \Rightarrow_{\mathcal{C}} \theta^\sharp(r)\ : c}$$

$$\text{(Fun)}\frac{\begin{array}{c}f : (\overline{a_1} \to b_1), \cdots, (\overline{a_k} \to b_k) \to c \in \Sigma \\ \rhd \Gamma, \overline{x_i : a_i} \vdash\ t_i \Rightarrow_{\mathcal{C}} t_i'\ : b_i \quad (\text{some } i \text{ s.t. } 1 \le i \le k)\end{array}}{\rhd \Gamma \vdash\ f(\overline{x_1^{a_1}}.t_1, \ldots, \overline{x_i^{a_i}}.t_i \ldots, \overline{x_k^{a_k}}.t_k) \Rightarrow_{\mathcal{C}} f(\overline{x_1^{a_1}}.t_1, \ldots, \overline{x_i^{a_i}}.t_i' \ldots, \overline{x_k^{a_k}}.t_k)\ : c}$$

Fig. 3. Second-order computation (one-step)

A meta-term t is called *well-typed* if $Z \rhd \Gamma \vdash\ t\ : c$ is derived by the typing rules in Fig. 2 for some Z, Γ, c.

The notation $t\,\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ denotes ordinary capture avoiding substitution that replaces the variables with terms $s_1, \ldots, s_n$.

Computation rules. For meta-terms $Z \rhd\ \vdash\ \ell\ : b$ and $Z \rhd\ \vdash\ r\ : b$ using a signature Σ, a *computation rule* is of the form $Z \rhd\ \vdash\ \ell \Rightarrow r\ : b$ satisfying:

(i) ℓ is a function term and a Miller's ***second-order pattern*** i.e., a meta-term in which every occurrence of meta-application is of the form $M[x_1, \ldots, x_n]$, where $x_1, \ldots, x_n$ are distinct bound variables.
(ii) all metavariables in r appear in ℓ.

Note that ℓ and r are meta-terms without free variables, but may have free ***meta***variables. A ***computation system*** (CS) is a pair $(\Sigma,\ \mathcal{C})$ of a signature and a set $\mathcal{C}$ of computation rules consisting of Σ-meta-terms. We write $s \Rightarrow_{\mathcal{C}} t$ to be one-step computation using $\mathcal{C}$ obtained by the inference system given in Fig. 3. We may omit some contexts and type information of a judgment, and simply write it as $Z \rhd \ell \Rightarrow r : b$, $\ell \Rightarrow_{\mathcal{C}} r$, or $\ell \Rightarrow r$ if they are clear from the context. From the viewpoint of pattern matching, (Rule) means that a computation system uses the decidable second-order pattern matching for one-step computation (cf. [Ham19, Sec.7.1]) not just syntactic matching. We regard $\Rightarrow_{\mathcal{C}}$ to be a binary relation on terms.

Example A2 The simply-typed λ-terms on the set LamTy of simple types generated by a set of base types BTy[6] are modeled in our setting as follows. We suppose type constructors L, Arr. The set of LamTy of all simple types for the λ-calculus is the least set satisfying

$$\mathrm{LamTy} = \mathrm{BTy} \cup \{\mathsf{Arr}(a, b) \mid a, b \in \mathrm{LamTy}\}.$$

[6] This is the set of all base types of the object-level simply-typed λ-calculus we will formulate, which should not be confused with a set $\mathcal{A}$ of atomic types of second-order computation system (at the meta-level).

$e ::=$	Expression
$\mid\ n$	Variable
$\mid\ \mathbf{lit}$	Literal
$\mid\ e_1\ e_2$	Application
$\mid\ \lambda n.e$	Abstraction
$\mid\ \mathbf{let}\ \mathit{binding}\ \mathbf{in}\ e$	Variable binding
$\mid\ \mathbf{case}\ e\ \mathbf{as}\ n\ \mathbf{return}\ \tau\ \mathbf{of}\ \overline{alt_i}^{\,i}$	Pattern match
$\mid\ e \rhd \gamma$	Cast
$\mid\ e_{\{tick\}}$	Internal note
$\mid\ \tau$	Type
$\mid\ \gamma$	Coercion

Fig. 4. The syntax of GHC Core

$[\![n]\!]$	$= \overline{x_i}^{\,i}.N[\overline{x_i}^{\,i}]$	if n has type $\overline{\tau_i}^{\,i} \to \tau$ and is a pattern variable
$[\![n]\!]$	$= x$	if n is not a pattern variable and $str(n) = x$
$[\![\mathbf{lit}]\!]$	$= lit\ ()$	if $str(\mathbf{lit}) = lit$
$[\![((e\ e_1)\ e_2)\dots e_n]\!]$	$= M[t_1,\dots,t_n]$	if e is a pattern variable and $[\![e]\!] = M$ and $[\![e_i]\!] = t_i$
$[\![((e\ e_1)\ e_2)\dots e_n]\!]$	$= f(t_1,\dots,t_n)$	if e is not a pattern variable and $[\![e]\!] = f$ and $[\![e_i]\!] = t_i$
$[\![\lambda\ n.e]\!]$	$= x.t$	if $str(n) = x$ and $[\![\mathrm{e}]\!] = \mathrm{t}$
$[\![e \rhd \gamma]\!]$	$= [\![e]\!]$	
$[\![e_{\{tick\}}]\!]$	$= [\![e]\!]$	

The function str gives the string representation.

Fig. 5. Translation from GHC Core expressions to SOL Terms

We use the mol type $\mathsf{L}(a)$ for encoding λ-terms of type $a \in \mathrm{LamTy}$. The λ-terms are given by a signature

$$\Sigma_{\mathrm{stl}} = \left\{ \begin{array}{l} \mathsf{lam}_{a,b} : (\mathsf{L}(a) \to \mathsf{L}(b)) \to \mathsf{L}(\mathsf{Arr}(a,b)) \\ \mathsf{app}_{a,b} : \mathsf{L}(\mathsf{Arr}(a,b)), \mathsf{L}(a) \to \mathsf{L}(b) \end{array} \ \middle|\ a,b \in \mathrm{LamTy} \right\}$$

The β-reduction law is presented as

$$\texttt{(beta)}\ \ M : \mathsf{L}(a) \to \mathsf{L}(b),\ N : \mathsf{L}(a)\ \rhd\ \vdash \mathsf{app}_{a,b}(\mathsf{lam}_{a,b}(x^a.\,M[x]),\ N)\ \Rightarrow\ M[N] : \mathsf{L}(b)$$

Note that $\mathsf{L}(\mathsf{Arr}(a,b))$ is a mol type, but $a \to b$ is not a mol type.

We use the following notational convention throughout the paper. We will present a signature by omitting mol type subscripts a, b. For example, simply writing function symbols lam and app, we mean $\mathsf{lam}_{a,b}$ and $\mathsf{app}_{a,b}$ in Σ_{stl} having appropriate mol type subscripts a, b.

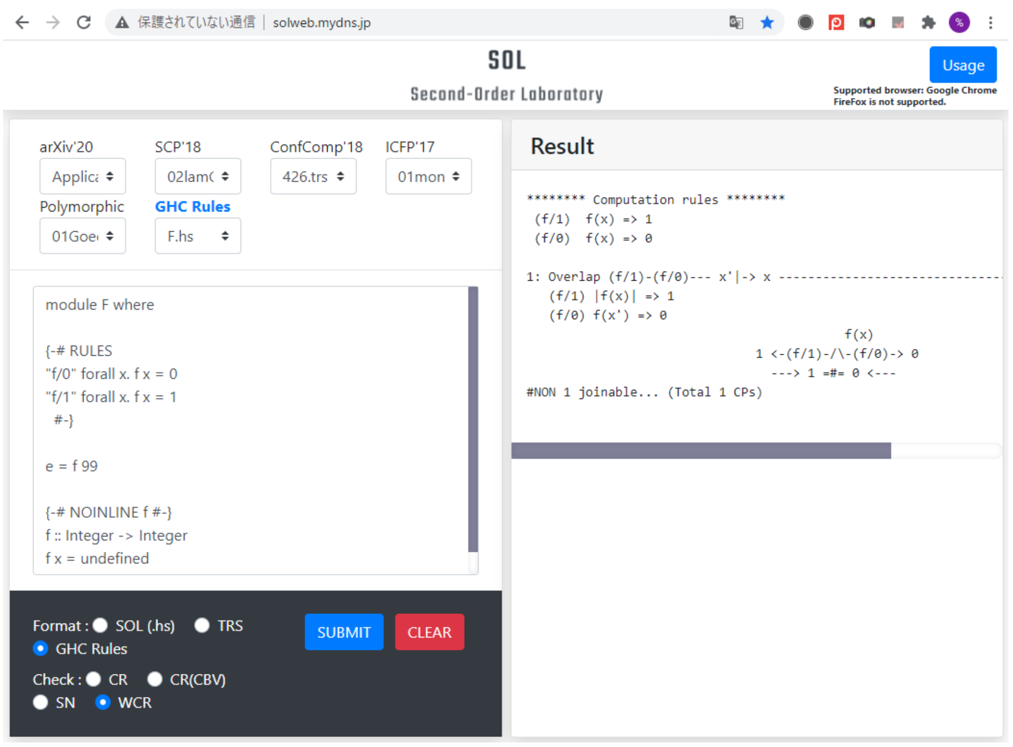
B From GHC Core Expressions to SOL Terms

Fig. 5 gives the translation function from GHC Core expressions to meta-terms.

```
 ******** Computation rules ********
(left/arr)  left(arr(x1.f[x1])) => arr(left(x1.f[x1]))
(right/arr)  right(arr(x1.f[x1])) => arr(right(x1.f[x1]))
(sum/arr)  arr(x1.f[x1]) +++ arr(x1.g[x1]) => arr(x1.f[x1] +++ x1.g[x1])
(fanin/arr)  arr(x1.f[x1]) ||| arr(x1.g[x1]) => arr(x1.f[x1] ||| x1.g[x1])
(compose/left)  left(f) . left(g) => left(f . g)
(compose/right)  right(f) . right(g) => right(f . g)
(compose/arr)  arr(x1.f[x1]) . arr(x1.g[x1]) => arr(x1.f[x1] . x1.g[x1])
(first/arr)  first(arr(x1.f[x1])) => arr(first(x1.f[x1]))
(second/arr)  second(arr(x1.f[x1])) => arr(second(x1.f[x1]))
(product/arr)  arr(x1.f[x1]) *** arr(x1.g[x1]) => arr(x1.f[x1] *** x1.g[x1])
(fanout/arr)  arr(x1.f[x1]) &&& arr(x1.g[x1]) => arr(x1.f[x1] &&& x1.g[x1])
(compose/first)  first(f) . first(g) => first(f . g)
(compose/second)  second(f) . second(g) => second(f . g)

1: Overlap (compose/left)-(left/arr)--- f|-> arr(x1'.f'[x1']) -------------------------------
   (compose/left) .(|left(f)|,left(g)) => left(f . g)
   (left/arr) left(arr(x1'.f'[x1'])) => arr(left(x1'.f'[x1']))
                          left(arr(x1'.f'[x1'])) . left(g)
left(arr(x1'.f'[x1']) . g) <-(compose/left)-/\-(left/arr)-> arr(left(x1d1.f'[x1d1])) . left(g)
        ---> left(arr(x1'.f'[x1']) . g) =#= arr(left(x1d1.f'[x1d1])) . left(g) <---
...
5: Overlap (compose/first)-(first/arr)--- f|-> arr(x1'.f'[x1']) -----------------------------
   (compose/first) .(|first(f)|,first(g)) => first(f . g)
   (first/arr) first(arr(x1'.f'[x1'])) => arr(first(x1'.f'[x1']))
                         first(arr(x1'.f'[x1'])) . first(g)
first(arr(x1'.f'[x1']) . g) <-(compose/first)-/\-(first/arr)-> arr(first(x1d1.f'[x1d1])) . first(g)
       ---> first(arr(x1'.f'[x1']) . g) =#= arr(first(x1d1.f'[x1d1])) . first(g) <---
6: Overlap (compose/first)-(first/arr)--- g|-> arr(x1'.f'[x1']) -----------------------------
   (compose/first) .(first(f),|first(g)|) => first(f . g)
   (first/arr) first(arr(x1'.f'[x1'])) => arr(first(x1'.f'[x1']))
                         first(f) . first(arr(x1'.f'[x1']))
first(f . arr(x1'.f'[x1'])) <-(compose/first)-/\-(first/arr)-> first(f) . arr(first(x1d1.f'[x1d1]))
       ---> first(f . arr(x1'.f'[x1'])) =#= first(f) . arr(first(x1d1.f'[x1d1])) <---
...
#NON 8 joinable... (Total 8 CPs)
```

Fig. 6. GSOL output: Critical pair checking of rules in `Control.Arrow`



```
module F where

{-# RULES
"f/0" forall x. f x = 0
"f/1" forall x. f x = 1
 #-}

e = f 99

{-# NOINLINE f #-}
f :: Integer -> Integer
f x = undefined
```

Format : SOL (.hs) TRS [✓ GHC Rules]

Check : CR CR(CBV) SN [✓ WCR]

Result

```
******** Computation rules ********
 (f/1)  f(x) => 1
 (f/0)  f(x) => 0

1: Overlap (f/1)-(f/0)--- x'|-> x -------------------------------
   (f/1) |f(x)| => 1
   (f/0) f(x') => 0
                                        f(x)
                                1 <-(f/1)-/\-(f/0)-> 0
                                  ---> 1 =#= 0 <---
#NON 1 joinable... (Total 1 CPs)
```

Fig. 7. GSOL web interface