Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order [*]

Martin Avanzini Ugo Dal Lago
Università di Bologna, Italy & INRIA, France
martin.avanzini@uibk.ac.at ugo.dallago@unibo.it

Georg Moser
University of Innsbruck, Austria
georg.moser@uibk.ac.at

Abstract

We show how the complexity of *higher-order* functional programs can be analysed automatically by applying program transformations to a defunctionalised versions of them, and feeding the result to existing tools for the complexity analysis of *first-order term rewrite systems*. This is done while carefully analysing complexity preservation and reflection of the employed transformations such that the complexity of the obtained term rewrite system reflects on the complexity of the initial program. Further, we describe suitable strategies for the application of the studied transformations and provide ample experimental data for assessing the viability of our method.

Categories and Subject Descriptors F.3.2 [*Semantics of programming languages*]: Program Analysis

Keywords Defunctionalisation, term rewriting, termination and resource analysis

1. Introduction

Automatically checking programs for correctness has attracted the attention of the computer science research community since the birth of the discipline. Properties of interest are not necessarily functional, however, and among the non-functional ones, noticeable cases are bounds on the amount of resources (like time, memory and power) programs need when executed.

Deriving upper bounds on the resource consumption of programs is indeed of paramount importance in many cases, but becomes undecidable as soon as the underlying programming language is non-trivial. If the units of measurement become concrete and close to the physical ones, the problem gets even more complicated, given the many transformation and optimisation layers programs are applied to before being executed. A typical example is the one of WCET techniques adopted in real-time systems [54], which do not only need to deal with how many machine instructions a program corresponds to, but also with how much time each instruction costs when executed by possibly complex architectures (including caches, pipelining, etc.), a task which is becoming even harder with the current trend towards multicore architectures.

As an alternative, one can analyse the *abstract* complexity of programs. As an example, one can take the number of instructions executed by the program or the number of evaluation steps to normal form, as a measure of its execution time. This is a less informative metric, which however becomes accurate if the actual time complexity *of each instruction* is kept low. One advantage of this analysis is the independence from the specific hardware platform executing the program at hand: the latter only needs to be analysed once. This is indeed a path which many have followed in the programming language community. A variety of verification techniques have been employed in this context, like abstract interpretations, model checking, type systems, program logics, or interactive theorem provers; see [3, 5, 35, 50] for some pointers. If we restrict our attention to higher-order functional programs, however, the literature becomes much sparser.

Conceptually, when analysing the time complexity of higher-order programs, there is a fundamental trade-off to be dealt with. On the one hand, one would like to have, at least, a clear relation between the cost attributed to a program and its actual complexity when executed: only this way the analysis' results would be informative. On the other hand, many choices are available as for how the complexity of higher-order programs can be evaluated, and one would prefer one which is closer to the programmer's intuitions. Ideally, then, one would like to work with an informative, even if not-too-concrete, cost measure, and to be able to evaluate programs against it fully automatically.

In recent years, several advances have been made such that the objectives above look now more realistic than in the past, at least as far as functional programming is concerned. First of all, some positive, sometime unexpected, results about the invariance of unitary cost models[1] have been proved for various forms of rewrite systems, including the λ-calculus [1, 6, 19]. What these results tell us is that counting the number of evaluation steps does *not* mean underestimating the time complexity of programs, which is shown to be bounded by a polynomial (sometime even by a linear function [2]) in their unitary cost. This is good news, since the number of rewrite steps is among the most intuitive notions of cost for functional programs, at least when time is the resource one is interested in.

But there is more. The rewriting-community has recently developed several tools for the automated time complexity analysis of *term rewrite system*, a formal model of computation that is at the heart of functional programming. Examples are `AProVE` [26], CaT [55], and TCT [8]. These *first-order provers* (FOPs for short) combine many different techniques, and after some years of development,

[*] This work was partially supported by FWF project number J3563, FWF project number P25781-N15 and by ANR project 14CE250005 ELICA.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada.
Copyright © 2015 ACM 978-1-4503-3669-7/15/08...$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] In the unitary cost model, a program is attributed a cost equal to the number of rewrite steps needed to turn it to normal form.

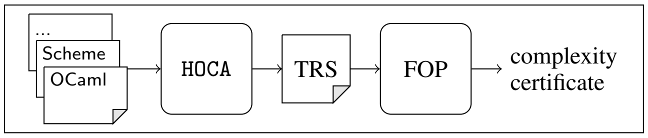

Figure 1. Complexity Analysis by HOCA and FOPs.

start being able to treat non-trivial programs, as demonstrated by the result of the annual termination competition.[2] This is potentially very interesting also for the complexity analysis of *higher-order functional programs*, since well-known transformation techniques such as *defunctionalisation* [48] are available, which turn higher-order functional programs into equivalent first-order ones. This has been done in the realm of termination [25, 44], but appears to be infeasible in the context of complexity analysis. Conclusively this program transformation approach has been reflected critical in the literature, cf. [35].

A natural question, then, is whether time complexity analysis of higher-order programs can indeed be performed by going through first-order tools. Is it possible to evaluate the unitary cost of functional programs by translating them into first-order programs, analysing them by existing first-order tools, and thus obtaining meaningful and informative results? Is, for example plain defunctionalisation enough? In this paper, we show that the questions above can be answered positively, when the necessary care is taken. We summarise the contributions of this paper.

1. We show how defunctionalisation is crucially employed in a transformation from higher-order programs to first-order term rewrite systems, such that the time complexity of the latter reflects upon the time complexity of the former. More precisely, we show a precise correspondence between the number of reduction steps of the higher-order program, and its defunctionalised version, represented as an *applicative term rewrite systems* (see Proposition 2).
2. But defunctionalisation is not enough. Defunctionalised programs have a recursive structure too complicated for FOPs to be effective on them. Our way to overcome this issue consists in further applying appropriate *program transformations*. These transformations must of course be proven correct to be viable. Moreover, we need the complexity analysis of the transformed program to mean something for the starting program, i.e. we also prove the considered transformations to be at least *complexity reflecting*, if not also *complexity preserving*. This addresses the problem that program transformations may potentially alter the resource usage. We establish *inlining* (see Corollary 1), *instantiation* (see Theorem 2), *uncurrying* (see Theorem 3), and *dead code elimination* (see Proposition 4) as, at least, complexity reflecting program transformations.
3. Still, analysing abstract program transformations is not yet sufficient. The main technical contribution of this paper concerns the *automation* of the program transformations rather than the abstract study presented before. In particular, automating instantiation requires dealing with the collecting semantics of the program at hand, a task we pursue by exploiting tree automata and control-flow analysis. Moreover, we define program transformation strategies which allow to turn complicated defunctionalised programs into simpler ones that work well in practice.
4. To evaluate our approach experimentally, we have built HOCA.[3] This tool is able to translate programs written in a pure, monomorphic subset of OCaml, into a first-order rewrite system, written in a format which can be understood by FOPs.

The overall flow of information is depicted in Figure 1. Note that by construction, the obtained certificate *reflects* onto the runtime complexity of the initial OCaml program, taking into account the standard semantics of OCaml. The figure also illustrates the *modularity* of the approach, as the here studied subset of OCaml just serves as a simple example language to illustrate the method: related languages can be analysed with the same set of tools, as long as the necessary transformation can be proven sound and complexity reflecting.

Our testbed includes standard higher-order functions like foldl and map, but also more involved examples such as an implementation of merge-sort using a higher-order divide-and-conquer combinator as well as simple parsers relying on the monadic parser-combinator outlined in Okasaki's functional pearl [43]. We emphasise that the methods proposed here are applicable in the context of non-linear runtime complexities. The obtained experimental results are quite encouraging.

The remainder of this paper is structured as follows. In the next section, we present our approach abstractly on a motivating example and clarify the challenges of our approach. In Section 3 we then present defunctionalisation formally. Section 4 presents the *transformation pipeline*, consisting of the above mentioned program transformations. Implementation issues and experimental evidence are given in Section 5 and 6, respectively. Finally, we conclude in Section 7, by discussing related work. An extended version of this paper with more details is available [10].

2. On Defunctionalisation: Ruling the Chaos

The main idea behind defunctionalisation is conceptually simple: function-abstractions are represented as first-order values; calls to abstractions are replaced by calls to a globally defined *apply-function*. Consider for instance the following OCaml-program:

```
let comp f g = fun z → f (g z) ;;
let rec walk xs =
  match xs with
    []  → (fun z → z)
  | x::ys  →  comp (walk ys)
              (fun z → x::z) ;;
let rev l = walk l [] ;;
let main l = rev l ;;
```

Given a list *xs*, the function walk constructs a function that, when applied to a list *ys*, appends *ys* to the list obtained by reversing *xs*. This function, which can be easily defined by recursion, is fed in rev with the empty list. The function main only serves the purpose of indicating the complexity of *which* function we are interested at.

Defunctionalisation can be understood already at this level. We first define a datatype for representing the three abstractions occurring in the program:

```
type 'a cl =
  Cl₁ of 'a cl * 'a cl  (* fun z → f (g z) *)
| Cl₂                   (* fun z → z *)
| Cl₃ of 'a             (* fun z → x::z *)
```

More precisely, an expression of type *'a cl* represents a function *closure*, whose arguments are used to store assignments to free variables. An infix operator (@), modelling application, can then be defined as follows:[4]

[2] http://termination-portal.org/wiki/Termination_Competition.

[3] Our tool HOCA is open source and available under http://cbr.uibk.ac.at/tools/hoca/.

[4] The definition is rejected by the OCaml type-checker, which however, is not an issue in our context.

```
let rec (@) cl z =
  match cl with
    Cl1(f,g)  →  f @ (g @ z)
  | Cl2  →  z
  | Cl3(x)  →  x::z ;;
```

Using this function, we arrive at a first-order version of the original higher-order function:

```
let comp f g = Cl1(f,g) ;;
let rec walk xs =
  match xs with
    []  →  Cl2
  | x::ys  →  comp (walk ys) Cl3(x) ;;
let rev l = walk l @ [] ;;
let main l = rev l ;;
```

Observe that now the recursive function `walk` constructs an explicit representation of the closure computed by its original definition. The function (`@`) carries out the remaining evaluation. This program can already be understood as a first-order rewrite system.

Of course, a systematic construction of the defunctionalized program requires some care. For instance, one has to deal with closures that originate from partial function applications. Still, the construction is quite easy to mechanize, see Section 3 for a formal treatment. On our running example, this program transformation results in the rewrite system $\mathcal{A}_{\mathsf{rev}}$, which looks as follows:[5]

```
1  Cl1(f,g) @ z  →  f @ (g @ z)
2  Cl2 @ z  →  z
3  Cl3(x) @ z  →  x::z
4  comp1(f) @ g  →  Cl1(f,g)
5  comp @ f  →  comp1(f)
6  match_walk([])  →  Cl2
7  match_walk(x::ys)  →
       comp @ (fix_walk @ ys) @ Cl3(x)
8  walk @ xs  →  match_walk(xs)
9  fix_walk @ xs  →  walk @ xs
10 rev @ l  →  fix_walk @ l @ []
11 main(l)  →  rev @ l
```

Despite its conceptual simplicity, current FOPs are unable to effectively analyse *applicative* rewrite systems, such as the one above. The reason this happens lies in the way FOPs work, which itself reflects the state of the art on formal methods for complexity analysis of first-order rewrite systems. In order to achieve composability of the analysis, the given system is typically split into smaller parts (see for example [9]), and each of them is analysed separately. Furthermore, contextualisation (aka *path analysis* [31]) and a restricted form of control flow graph analysis (or *dependency pair analysis* [30, 42]) is performed. However, at the end of the day, syntactic and semantic basic techniques, like path orders or interpretations [52, Chapter 6] are employed. All these methods focus on the analysis of the given defined symbols (like for instance the application symbol in the example above) and fail if their recursive definition is too complicated. Naturally this calls for a special treatment of the applicative structure of the system [32].

How could we get rid of those (`@`), thus highlighting the deep recursive structure of the program above? Let us, for example, focus on the rewriting rule

$$\mathtt{Cl}_1(f,g)\ \mathtt{@}\ z \rightarrow f\ \mathtt{@}\ (g\ \mathtt{@}\ z)\ ,$$

which is particularly nasty for FOPs, given that the variables f and g will be substituted by unknown functions, which could potentially have a very high complexity. How could we *simplify* all this? The key observation is that although this rule tells us how to compose two *arbitrary* closures, only very few instances of the rule above are needed, namely those were g is of the form $\mathtt{Cl}_3(x)$, and f is either $\mathtt{Cl}_2$ or again of the form $\mathtt{Cl}_1(f,g)$. This crucial information can be retrieved in the so-called *collecting semantics* [41] of the term rewrite system above, which precisely tells us which object will possibly be substituted for rule variables along the evaluation of certain families of terms. Dealing with all this fully automatically is of course impossible, but techniques based on tree automata, and inspired by those in [33] can indeed be of help.

Another useful observation is the following: function symbols like, e.g. `comp` or $\mathtt{match}_{\mathtt{walk}}$ are essentially useless: their only purpose is to build intermediate closures, or to control program flow: one could simply shortcircuit them, using a form of *inlining*. And after this is done, some of the left rules are *dead code*, and can thus be eliminated from the program. Finally we arrive at a truly first-order system and *uncurrying* brings it to a format most suitable for FOPs.

If we carefully apply the just described ideas to the example above, we end up with the following first-order system, called $\mathcal{R}_{\mathsf{rev}}$, which is precisely what `HOCA` produces in output:

```
Cl1^1(Cl2,Cl3(x),z)  →  x::z
Cl1^1(Cl1(f,g),Cl3(x),z)  →  Cl1^1(f,g,x::z)
fix_walk^1([])  →  Cl2
fix_walk^1(x:ys)  →  Cl1(fix_walk^1(ys),Cl3(x))
main(l)  →  []
main(x:ys)  →  Cl1^1(fix_walk^1(ys),Cl3(x),[])
```

This term rewrite system is equivalent to $\mathcal{A}_{\mathsf{rev}}$ from above, both extensionally and in terms of the underlying complexity. However, the FOPs we have considered can indeed conclude that `main` has linear complexity, a result that can be in general lifted back to the original program.

Sections 4 and 5 are concerned with a precise analysis of the program transformations we employed when turning $\mathcal{A}_{\mathsf{rev}}$ into $\mathcal{R}_{\mathsf{rev}}$. Before, we recap central definitions in the next section.

3. Preliminaries

The purpose of this section is to give some preliminary notions about the λ-calculus, term rewrite systems, and translations between them; see [11, 45, 52] for further reading.

To model a reasonable rich but pure and monomorphic functional language, we consider a typed λ-calculus with constants and fixpoints akin to Plotkin's PCF [46]. To seamlessly express programs over algebraic datatypes, we allow constructors and pattern matching. To this end, let $\mathtt{C}_1, \ldots, \mathtt{C}_k$ be finitely many constructors, each equipped with a fixed *arity*. The syntax of PCF-programs is given by the following grammar:

$$\begin{aligned} \mathsf{Exp} \quad e, f \ ::= \ & x \mid \mathtt{C}_i(\vec{e}) \mid \lambda x.e \mid e\ f \mid \mathsf{fix}(x.e) \\ & \mid \mathsf{match}\ e\ \{\mathtt{C}_1(\vec{x}_1) \mapsto e_1; \cdots ; \mathtt{C}_k(\vec{x}_k) \mapsto e_k\}\ , \end{aligned}$$

where x ranges over variables. Note that the variables $\vec{x}_i$ in a match-expression are considered bound in e_i. A simple type system can be easily defined [10] based on a single ground type, and on the usual arrow type constructor. We claim that extending the language with products and coproducts would not be problematic.

[5] In $\mathcal{A}_{\mathsf{rev}}$, rule (9) reflects that, under the hood, we treat recursive let expressions as syntactic sugar for a dedicated fixpoint operator.

We adopt *weak call-by-value* semantics. Here *weak* means that reduction under any λ-abstraction $\lambda x.e$ and any fixpoint-expressions $\mathsf{fix}(x.e)$ is prohibited. The definition is straightforward, see e.g. [29]. *Call-by-value* means that in a redex $e\ f$, the expression f has to be evaluated. A match-expression $\mathtt{match}\ e\ \{\mathtt{C}_1(\vec{x}_1) \mapsto e_1; \cdots ; \mathtt{C}_k(\vec{x}_k) \mapsto e_k\}$ is evaluated by first evaluating the guard e to a value $\mathtt{C}_i(\vec{v})$. Reduction then continues with the corresponding case-expression e_i with values $\vec{v}_i$ substituted for variables $\vec{x}_i$. The one-step weak call-by-value reduction relation is denoted by $\to_v$. Elements of the term algebra over constructors $\mathtt{C}_1, \dots, \mathtt{C}_k$ embedded in our language are collected in Input. A *PCF-program* with n *input arguments* is a closed expression $P = \lambda x_1 \cdots \lambda x_n.e$ of first-order type. What this implicitly means is that we are interested in an analysis of programs with a possibly very intricate internal higher-order structure, but whose arguments are values of ground type. In this is akin to the setting in [12] and provides an intuitive notion of runtime complexity for higher-order programs, without having to rely on ad-hoc restrictions on the use of function-abstracts (as e.g. [35]). This way we also ensure that the abstractions reduced in a run of P are the ones found in P, an essential property for performing defunctionalisation. We assume that variables in P have been renamed apart, and we impose a total order on variables in P. The free variables $\mathsf{FV}(e)$ in the body e of P can this way be defined as an ordered sequence of variables.

Example 1. We fix constructors `[]` and `(::)` for lists, the latter we write infix. Then the program computing the reverse of a list, as described in the previous section, can be seen as the PCF term $P_{\mathsf{rev}} := \lambda l.\mathit{rev}\ l$ where

$$\begin{aligned}
\mathit{rev} &= \lambda l.\mathsf{fix}(w.\mathit{walk})\ l\ \mathtt{[]}\ ;\\
\mathit{walk} &= \lambda xs.\mathtt{match}\ xs\ \left\{\begin{array}{l} \mathtt{[]} \mapsto \lambda z.z\ ;\\ x\mathtt{::}ys \mapsto \mathit{comp}\ (w\ ys)\ (\lambda z.x\mathtt{::}z)\end{array}\right\};\\
\mathit{comp} &= \lambda f.\lambda g.\lambda z.f\ (g\ z)\ .
\end{aligned}$$

The second kind of programming formalism we will deal with is the one of *term rewrite systems* (TRSs for short). Let $\mathcal{F} = \{\mathtt{f}_1, \dots, \mathtt{f}_n\}$ be a set of function symbols, each equipped again with an arity, the *signature*. We denote by $s, t, \dots$ terms over the signature $\mathcal{F}$, possibly including variables. A *position* p in t is a finite sequence of integers, such that the following definition of *subterm at position* p is well-defined: $t|_\epsilon = t$ for the *empty position* ϵ, and $t|_{ip} = t_i|_p$ for $t = \mathtt{f}(t_1, \dots, t_k)$. For a position p in t, we denote by $t[s]_p$ the term obtained by replacing the subterm at position p in t by the term s. A *context* C is a term containing one occurrence of a special symbol $\Box$, the *hole*. We define $C[t] := C[t]_p$ for p the position of $\Box$ in C, i.e. $C|_p = \Box$.

A *substitution*, is a finite mapping σ from variables to terms. By $t\sigma$ we denote the term obtained by replacing in t all variables x in the domain of σ by $\sigma(x)$. A substitution σ is *at least as general* as a substitution τ if there exists a substitution ρ such that $\tau(x) = \sigma(x)\rho$ for each variable x. A term t is an instance of a term s if there exists a substitution σ, with $s\sigma = t$; the terms t and s *unify* if there exists a substitution μ, the *unifier*, such that $t\mu = s\mu$. If two terms are unifiable, then there exists a *most general unifier* (mgu for short).

A *term rewrite system* $\mathcal{R}$ is a finite set of rewrite rules, i.e. directed equations $\mathtt{f}(l_1, \dots, l_k) \to r$ such that all variables occurring in the *right-hand side* r occur also in the *left-hand side* $\mathtt{f}(l_1, \dots, l_k)$. The roots of left-hand sides, the *defined symbols* of $\mathcal{R}$, are collected in $\mathcal{D}_\mathcal{R}$, the remaining symbols $\mathcal{F} \setminus \mathcal{D}_\mathcal{R}$ are the *constructors* of $\mathcal{R}$ and collected in $\mathcal{C}_\mathcal{R}$. Terms over the constructors $\mathcal{C}_\mathcal{R}$ are considered *values* and collected in $\mathcal{T}(\mathcal{C}_\mathcal{R})$. We denote by $\to_\mathcal{R}$ the one-step rewrite relation of $\mathcal{R}$, imposing *call-by-value* semantics. Call-by-value means that variables are assigned elements of $\mathcal{T}(\mathcal{C}_\mathcal{R})$. Throughout the following, we consider *non-ambiguous* rewrite systems, that is, the left-hand sides are pairwise *non-overlapping*. Despite the restriction to non-ambiguous rewrite systems, the rewrite relation $\to_\mathcal{R}$ may be non-deterministic: e.g. no control in what order arguments of terms are reduced is present. However, the following special case of the parallel moves lemma [11] tells us that this form of non-determinism is not harmful for complexity-analysis.

Proposition 1. *For a non-ambiguous TRS $\mathcal{R}$, all* normalising *reductions of t have the same length, i.e. if $t \to^m_\mathcal{R} u_1$ and $t \to^n_\mathcal{R} u_2$ for two irreducible terms u_1 and u_2, then $u_1 = u_2$ and $m = n$.*

An *applicative term rewrite system* (*ATRS* for short) is usually defined as a TRS over a signature consisting of a finite set of nullary function symbols and one dedicated binary symbol $(\mathtt{@})$, the *application symbol*. We follow the usual convention that $(\mathtt{@})$ associates to the left. Here, we are more liberal and just assume the presence of $(\mathtt{@})$, and allow function symbols that take more than one argument. Throughout the following, we are foremost dealing with ATRSs, which we denote by $\mathcal{A}, \mathcal{B}$ below. We also write $(\mathtt{@})$ infix and assume that it associates to the left.

In the following, we show that every PCF-program P can be seen as an ATRS $\mathcal{A}_P$. To this end, we first define an *infinite schema* $\mathcal{A}_{\mathsf{PCF}}$ of rewrite rules which allows us to evaluate the whole of PCF. The signature underlying $\mathcal{A}_{\mathsf{PCF}}$ contains, besides the application-symbol $(\mathtt{@})$ and constructors $\mathtt{C}_1, \dots, \mathtt{C}_k$, the following function symbols, called *closure constructors*: (i) for each PCF term $\lambda x.e$ with n free variables an n-ary symbol $\mathtt{lam}_{x.e}$; (ii) for each PCF term $\mathsf{fix}(x.e)$ with n free variables an n-ary symbol $\mathtt{fix}_{x.e}$; and (iii) for each match-expression $\mathtt{match}\ e\ \{cs\}$ with n free variables a symbol $\mathtt{match}_{cs}$ of arity $n+1$. Furthermore, we define a mapping $[\,\cdot\,]_\Phi$ from PCF terms to $\mathcal{A}_{\mathsf{PCF}}$ terms as follows.

$$\begin{aligned}
[\,x\,]_\Phi &:= x\ ;\\
[\,\lambda x.e\,]_\Phi &:= \mathtt{lam}_{x.e}(\vec{x}), \text{ where } \vec{x} = \mathsf{FV}(\lambda x.e)\ ;\\
[\,\mathtt{C}_i(e_1, \dots, e_k)\,]_\Phi &:= \mathtt{C}_i([\,e_1\,]_\Phi, \dots, [\,e_k\,]_\Phi)\ ;\\
[\,e\ f\,]_\Phi &:= [\,e\,]_\Phi \mathbin{\mathtt{@}} [\,f\,]_\Phi\ ;\\
[\,\mathsf{fix}(x.e)\,]_\Phi &:= \mathtt{fix}_{x.e}(\vec{x}), \text{ where } \vec{x} = \mathsf{FV}(\mathsf{fix}(x.e))\ ;\\
[\,\mathtt{match}\ e\ \{cs\}\,]_\Phi &:= \mathtt{match}_{cs}([\,e\,]_\Phi, \vec{x}), \text{ where } \vec{x} = \mathsf{FV}(\{cs\})\ .
\end{aligned}$$

Based on this interpretation, each closure constructor is equipped with one or more of the following *defining rules*:

$$\begin{aligned}
&\mathtt{lam}_{x.e}(\vec{x}) \mathbin{\mathtt{@}} x \to [\,e\,]_\Phi\ ;\\
&\mathtt{fix}_{x.e}(\vec{x}) \mathbin{\mathtt{@}} y \to [\,e\{\mathsf{fix}(x.e)/x\}\,]_\Phi \mathbin{\mathtt{@}} y\ , \text{ where } y \text{ is fresh;}\\
&\mathtt{match}_{cs}(\mathtt{C}_i(\vec{x}_i), \vec{x}) \to [\,e_i\,]_\Phi, \text{ for } i \in \{1, \dots, k\}.
\end{aligned}$$

Here, we suppose $cs = \{\mathtt{C}_1(\vec{x}_1) \mapsto e_1; \cdots ; \mathtt{C}_k(\vec{x}_k) \mapsto e_k\}$.

For a program $P = \lambda x_1 \cdots \lambda x_n.e$, we define $\mathcal{A}_P$ as the least set of rules that (i) contains a rule $\mathtt{main}(x_1, \dots, x_n) \to [\,e\,]_\Phi$, where $\mathtt{main}$ is a dedicated function symbol; and (ii) whenever $l \to r \in \mathcal{A}_P$ and $\mathtt{f}$ is a closure-constructor in r, then $\mathcal{A}_P$ contains all defining rules of $\mathtt{f}$ from the schema $\mathcal{A}_{\mathsf{PCF}}$. Crucial, $\mathcal{A}_P$ is always finite, in fact, the size of $\mathcal{A}_P$ is linearly bounded in the size of P [10].

Remark. This statement becomes trivial if we consider alternative defining rule

$$\mathtt{fix}_{x.e}(\vec{x}) \mathbin{\mathtt{@}} y \to [\,e\,]_\Phi\{x/\mathtt{fix}_{x.e}(\vec{x})\} \mathbin{\mathtt{@}} y\ ,$$

which would also correctly model the semantics of fixpoints $\mathsf{fix}(x.e)$. Then the closure constructors occurring in $\mathcal{A}_P$ are all obtained from sub-expressions of P. Our choice is motivated by the fact that closure constructors of fixpoints are propagates to call sites, something that facilitates the complexity analysis of $\mathcal{A}_P$.

Example 2. The expression P_{rev} from Example 1 gets translated into the ATRS $\mathcal{A}_{P_{\mathsf{rev}}} = \mathcal{A}_{\mathsf{rev}}$ we introduced in Section 2. For readability, closure constructors have been renamed.

We obtain the following simulation result, a proof of which can be found in [10].

Proposition 2. *Every $\to_v$-reduction of an expression $P\ d_1\ \cdots\ d_n$ ($d_j \in \mathsf{Input}$) is simulated step-wise by a call-by-value $\mathcal{A}_P$-derivation starting from $\mathtt{main}(d_1, \ldots, d_n)$.*

As the inverse direction of this proposition can also be stated, $\mathcal{A}_P$ can be seen as a sound and complete, in particular step-preserving, implementation of the PCF-program P.

In correspondence to Proposition 2, we define the runtime complexity of an ATRS $\mathcal{A}$ as follows. As above, only terms $d \in \mathsf{Input}$ built from the constructors $\mathcal{C}$ are considered valid inputs. The *runtime of $\mathcal{A}$ on inputs* $d_1, \ldots, d_n$ is defined as the length of the longest rewrite sequence starting from $\mathtt{main}(d_1, \ldots, d_n)$. The *runtime complexity function* is defined as the (partial) function which maps the natural number m to the maximum runtime of $\mathcal{A}$ on inputs $d_1, \ldots, d_n$ with $\sum_{j=1}^{n} |d_j| \leqslant m$, where the *size* $|d|$ is defined as the number of occurrences of constructors in d.

Crucial, our notion of runtime complexity corresponds to the notion employed in first-order rewriting and in particular in FOPs. Our simple form of defunctionalisation thus paves the way to our primary goal: a successful complexity analysis of $\mathcal{A}_P$ with rewriting-based tools can be relayed back to the PCF-program P.

4. Complexity Reflecting Transformations

The result offered by Proposition 2 is remarkable, but is a Pyrrhic victory towards our final goal: as discussed in Section 2, the complexity of defunctionalised programs is hard to analyse, at least if one wants to go via FOPs. It is then time to introduce the four program transformations that form our toolbox, and that will allow us to turn defunctionalised programs into ATRSs which are easier to analyse.

In this section, we describe the four transformations abstractly, without caring too much about *how* one could implement them. Rather, we focus on their correctness and, even more importantly for us, we verify that the complexity of the transformed program is not too small compared to the complexity of the original one. We will also show, through examples, how all this can indeed be seen as a way to simplify the recursive structure of the programs at hand.

A *transformation* is a partial function f from ATRSs to ATRSs. In the case that $f(\mathcal{A})$ is undefined, the transformation is called *inapplicable* to $\mathcal{A}$. We call the transformation f (*asymptotically*) *complexity reflecting* if for every ATRS $\mathcal{A}$, the runtime complexity of $\mathcal{A}$ is bounded (asymptotically) by the runtime complexity of $f(\mathcal{A})$, whenever f is applicable on $\mathcal{A}$. Conversely, we call f (*asymptotically*) *complexity preserving* if the runtime complexity of $f(\mathcal{A})$ is bounded (asymptotically) by the complexity of $\mathcal{A}$, whenever f is applicable on $\mathcal{A}$. The former condition states a form of *soundness:* if f is complexity reflecting, then a bound on the runtime complexity of $f(\mathcal{A})$ can be relayed back to $\mathcal{A}$. The latter conditions states a form of *completeness*: application of a complexity preserving transformation f will not render our analysis ineffective, simply because f translated $\mathcal{A}$ to an inefficient version. We remark that the set of complexity preserving (complexity reflecting) transformations is closed under composition.

4.1 Inlining

Our first transformation constitutes a form of *inlining*. This allows for the elimination of auxiliary functions, this way making the recursive structure of the considered program apparent.

Consider the ATRS $\mathcal{A}_{\mathsf{rev}}$ from Section 2. There, for instance, the call to $\mathtt{walk}$ in the definition of $\mathtt{fix}_{\mathtt{walk}}$ could be *inlined*, thus resulting in a new definition:

$$\mathtt{fix}_{\mathtt{walk}} \mathbin{@} xs \ \to\ \mathtt{match}_{\mathtt{walk}}(xs)\ .$$

Informally, thus, inlining consists in modifying the right-hand-sides of ATRS rules by rewriting subterms, according to the ATRS itself. We will also go beyond rewriting, by first specializing arguments so that a rewrite triggers. In the above rule for instance, $\mathtt{match}_{\mathtt{walk}}$ cannot be inlined immediately, simply because $\mathtt{match}_{\mathtt{walk}}$ is defined itself by case analysis on xs. To allow inlining of this function nevertheless, we specialize xs to the patterns $\mathtt{[]}$ and $x\mathtt{::}ys$, the patterns underlying the case analysis of $\mathtt{match}_{\mathtt{walk}}$. This results in two alternative rules for $\mathtt{fix}_{\mathtt{walk}}$, namely

$$\mathtt{fix}_{\mathtt{walk}} \mathbin{@} \mathtt{[]} \to \mathtt{match}_{\mathtt{walk}}(\mathtt{[]})\ ;$$
$$\mathtt{fix}_{\mathtt{walk}} \mathbin{@} (x\mathtt{::}ys) \to \mathtt{match}_{\mathtt{walk}}(x\mathtt{::}ys)\ .$$

Now we can inline $\mathtt{match}_{\mathtt{walk}}$ in both rules. As a consequence the rules defining $\mathtt{fix}_{\mathtt{walk}}$ are easily seen to be structurally recursive, a fact that FOPs can recognise and exploit.

A convenient way to formalise inlining is by way of *narrowing* [11]. We say that a term s *narrows* to a term t at a non-variable position p in s, in notation $s \overset{\mu}{\rightsquigarrow}_{\mathcal{A},p} t$, if there exists a rule $l \to r \in \mathcal{A}$ such that μ is a unifier of left-hand side l and the subterm $s|_p$ (after renaming apart variables in $l \to r$ and s) and $t = s\mu[r\mu]_p$. In other words, the instance $s\mu$ of s rewrites to t at position p with rule $l \to r \in \mathcal{A}$. The substitution μ is just enough to uncover the corresponding redex in s. Note, however, that the performed rewrite step is not necessarily call-by-value, the mgu μ could indeed contain function calls. We define the set of all *inlinings* of a rule $l \to r$ at position p which is labeled by a defined symbol by

$$\mathsf{inline}_{\mathcal{A},p}(l \to r) := \{l\mu \to s \mid r \overset{\mu}{\rightsquigarrow}_{\mathcal{A},p} s\}\ .$$

The following example demonstrates inlining through narrowing.

Example 3. Consider the substitutions $\mu_1 = \{xs \mapsto \mathtt{[]}\}$ and $\mu_2 = \{xs \mapsto x\mathtt{::}ys\}$. Then we have

$$\mathtt{match}_{\mathtt{walk}}(xs) \overset{\mu_1}{\rightsquigarrow}_{\mathcal{A}_{\mathsf{rev}},\epsilon} \mathtt{Cl}_2$$
$$\mathtt{match}_{\mathtt{walk}}(xs) \overset{\mu_2}{\rightsquigarrow}_{\mathcal{A}_{\mathsf{rev}},\epsilon} \mathtt{comp} \mathbin{@} (\mathtt{fix}_{\mathtt{walk}} \mathbin{@} ys) \mathbin{@} \mathtt{Cl}_3(x)\ .$$

Since no other rule of $\mathcal{A}_{\mathsf{rev}}$ unifies with $\mathtt{match}_{\mathtt{walk}}(xs)$, the set

$$\mathsf{inline}_{\mathcal{A}_{\mathsf{rev}},\epsilon}(\mathtt{fix}_{\mathtt{walk}} \mathbin{@} xs \ \to\ \mathtt{match}_{\mathtt{walk}}(xs))$$

consists of the two rules

$$\mathtt{fix}_{\mathtt{walk}} \mathbin{@} \mathtt{[]} \to \mathtt{Cl}_2\ ;$$
$$\mathtt{fix}_{\mathtt{walk}} \mathbin{@} (x\mathtt{::}ys) \to \mathtt{comp} \mathbin{@} (\mathtt{fix}_{\mathtt{walk}} \mathbin{@} ys) \mathbin{@} \mathtt{Cl}_3(x)\ .$$

Inlining is in general not complexity reflecting. Indeed, inlining is employed by many compilers as a program optimisation technique. The following examples highlight two issues we have to address. The first example indicates the obvious: in a call-by-value setting, inlining is not asymptotically complexity reflecting, if potentially expensive function calls in arguments are deleted.

Example 4. Consider the following inefficient system:

```
1 k(x,y)        →  x
2 main(0)       →  0
3 main(S(n))    →  k(main(n),main(n))
```

Inlining $\mathtt{k}$ in the definition of $\mathtt{main}$ results in an alternative definition $\mathtt{main}(\mathtt{S}(n)) \to \mathtt{main}(n)$ of rule (3), eliminating one of the two recursive calls and thereby reducing the complexity from exponential to linear.

The example motivates the following, easily decidable, condition. Let $l \to r$ denote a rule whose right-hand side is subject to inlining at position p. Suppose the rule $u \to v \in \mathcal{A}$ is unifiable with the subterm $r|_p$ of the right-hand side r, and let μ denote the most general unifier. Then we say that inlining $r|_p$ with $u \to v$ is *redex* *preserving* if whenever $x\mu$ contains a defined symbol of $\mathcal{A}$, then the variable x occurs also in the right-hand side v. The inlining $l \to r$ at position p is called *redex preserving* if inlining $r|_p$ is redex preserving with *all rule* $u \to v$ such that u unifies with $r|_p$. Redex-preservation thus ensures that inlining does not delete potential function calls, apart from the inlined one. In the example above, inlining `k(main(`n`),main(`n`))` is not redex preserving because the variable y is mapped to the redex `main(`n`)` by the underlying unifier, but y is deleted in the inlining rule `k(`x`,`y`)` $\to$ x.

Our second example is more subtle and arises when the studied rewrite system is under-specified:

Example 5. Consider the system consisting of the following rules.

```
1  h(x,0)      →  x
2  main(0)     →  0
3  main(S(n))  →  h(main(n),n)
```

Inlining `h` in the definition of `main` will specialise the variable n to `0` and thus replaces rule (3) by `main(S(0))` $\to$ `main(0)`. Note that the runtime complexity of the former system is linear, whereas its runtime complexity is constant after transformation.

Crucial for the example, the symbol `h` is not *sufficiently defined*, i.e. the computation gets stuck after completely unfolding `main`. To overcome this issue, we require that inlined functions are sufficiently defined. Here a defined function symbol `f` is called *sufficiently defined*, with respect to an ATRS $\mathcal{A}$, if all subterms $\mathtt{f}(\vec{t})$ occurring in a reduction of $\mathtt{main}(d_1, \ldots, d_n)$ ($d_j \in \mathsf{Input}$) are reducible. This property is not decidable in general. Still, the ATRSs obtained from the translation in Section 3 satisfy this condition for all defined symbols: by construction, reductions do not get stuck. Inlining, and the transformations discussed below, preserve this property.

We will now show that under the above outlined conditions, inlining is indeed complexity reflecting. Fix an ATRS $\mathcal{A}$. In proofs below, we denote by $\rightsquigarrow_{\mathcal{A}}$ an extension of $\to_{\mathcal{A}}$ where not all arguments are necessarily reduced, but where still a step cannot delete redexes: $s \rightsquigarrow_{\mathcal{A}} t$ if $s = C[l\sigma]$ and $t = C[r\sigma]$ for a context C, rule $l \to r \in \mathcal{A}$ and a substitution σ which satisfies $\sigma(x) \in \mathcal{T}(\mathcal{C}_{\mathcal{A}})$ for all variables x which occur in l but not in r. By definition, $\to_{\mathcal{A}} \subseteq \rightsquigarrow_{\mathcal{A}}$. The relation $\rightsquigarrow_{\mathcal{A}}$ is just enough to capture rewrites performed on right-hand sides in a complexity reflecting inlining.

The next lemma collects the central points of our correctness proof. Here, we first consider the effect of replacing a single application of a rule $l \to r$ with an application of a corresponding rule in $\mathsf{inline}_{\mathcal{A},p}(l \to r)$. As the lemma shows, this is indeed always possible, provided the inlined function is sufficiently defined. Crucial, inlining preserves semantics. Complexity reflecting inlining, on the other hand, does not optimise the ATRS under consideration too much, if at all.

Lemma 1. *Let $l \to r$ be a rewrite rule subject to a redex preserving inlining of function* `f` *at position p in r. Suppose that the symbol* `f` *is sufficiently defined by $\mathcal{A}$. Consider a normalising reduction*

$$\mathtt{main}(d_1, \ldots, d_n) \to^*_{\mathcal{A}} C[l\sigma] \to_{\mathcal{A}} C[r\sigma] \to^{\ell}_{\mathcal{A}} u\ ,$$

for $d_i \in \mathsf{Input}$ $(i = 1, \ldots, n)$ and some $\ell \in \mathbb{N}$. Then there exists a term t such that the following properties hold:

1. *$l\sigma \to_{\mathsf{inline}_{\mathcal{A},p}(l\to r)} t$; and*
2. *$r\sigma \rightsquigarrow_{\mathcal{I}} t$, where $\mathcal{I}$ collects all rules that are unifiable with the right-hand side r at position p; and*
3. *$C[t] \to^{\geqslant \ell - 1}_{\mathcal{A}} u$.*

In consequence, we thus obtain a term t such that

$$\mathtt{main}(d_1, \ldots, d_n) \to^*_{\mathcal{A}} C[l\sigma] \to_{\mathsf{inline}_{\mathcal{A},p}(l\to r)} C[t] \to^{\geqslant \ell-1}_{\mathcal{A}} u\ ,$$

holds under the assumptions of the lemma. Complexity preservation of inlining, modulo a constant factor under the outlined assumption, now follows essentially by induction on the maximal length of reductions. As a minor technical complication, we have to consider the broader reduction relation $\rightsquigarrow_{\mathcal{A}}$ instead of $\to_{\mathcal{A}}$. To ensure that the induction is well-defined, we use the following specialization of [28, Theorem 3.13].

Proposition 3. *If a term t has a normalform wrt. $\to_{\mathcal{A}}$, then* all $\rightsquigarrow_{\mathcal{A}}$ *reductions of t are finite.*

Theorem 1. *Let $l \to r$ be a rewrite rule subject to a redex preserving inlining of function* `f` *at position p in r. Suppose that the symbol* `f` *is sufficiently defined by $\mathcal{A}$. Let $\mathcal{B}$ be obtained by replacing rule $l \to r$ by the rules $\mathsf{inline}_{\mathcal{A},p}(l \to r)$. Then every normalizing derivation with respect to $\mathcal{A}$ starting from $\mathtt{main}(d_1, \ldots, d_n)$ ($d_j \in \mathsf{Input}$) of length ℓ is simulated by a derivation with respect to $\mathcal{B}$ from $\mathtt{main}(d_1, \ldots, d_n)$ of length at least $\lfloor \frac{\ell}{2} \rfloor$.*

Proof. Suppose t is a reduct of $\mathtt{main}(d_1, \ldots, d_n)$ occurring in a normalising reduction, i.e. $\mathtt{main}(d_1, \ldots, d_n) \to^*_{\mathcal{A}} t \to^*_{\mathcal{A}} u$, for u a normal form of $\mathcal{A}$. In proof, we show that if $t \to^*_{\mathcal{A}} u$ is a derivation of length ℓ, then there exists a normalising derivation with respect to $\mathcal{B}$ whose length is at least $\lfloor \frac{\ell}{2} \rfloor$. The theorem then follows by taking $t = \mathtt{main}(d_1, \ldots, d_n)$.

We define the *derivation height* $\mathrm{dh}(s)$ of a term s wrt. the relation $\rightsquigarrow_{\mathcal{A}}$ as the maximal m such that $t \rightsquigarrow^m_{\mathcal{A}} u$ holds. The proof is by induction on $\mathrm{dh}(t)$, which is well-defined by assumption and Proposition 3. It suffices to consider the induction step. Suppose $t \to_{\mathcal{A}} s \to^{\ell}_{\mathcal{A}} u$. We consider the case where the step $t \to_{\mathcal{A}} s$ is obtained by applying the rule $l \to r \in \mathcal{A}$, otherwise, the claim follows directly from induction hypothesis. Then as a result of Lemma 1(1) and 1(3) we obtain an alternative derivation

$$t \to_{\mathcal{B}} w \to^{\ell'}_{\mathcal{A}} u\ ,$$

for some term w and ℓ' satisfying $\ell' \geqslant \ell - 1$. Note that $s \rightsquigarrow_{\mathcal{A}} w$ as a consequence of Lemma 1(2), and thus $\mathrm{dh}(s) > \mathrm{dh}(w)$ by definition of derivation height. Induction hypothesis on w thus yields a derivation $t \to_{\mathcal{B}} w \to^*_{\mathcal{B}} u$ of length at least $\lfloor \frac{\ell'}{2} \rfloor + 1 = \lfloor \frac{\ell'+2}{2} \rfloor \geqslant \lfloor \frac{\ell+1}{2} \rfloor$. □

We can then obtain that inlining has the key property we require on transformations.

Corollary 1 (Inlining Transformation)**.** *The* inlining transformation, *which replaces a rule $l \to r \in \mathcal{A}$ by $\mathsf{inline}_{\mathcal{A},p}(l \to r)$, is asymptotically complexity reflecting whenever the function considered for inlining is sufficiently defined and the inlining itself is redex preserving.*

Example 6. Consider the ATRS $\mathcal{A}_{\mathsf{rev}}$ from Section 2. Three applications of inlining result in the following ATRS:

```
Cl₁(f,g) @ z  →  f @ (g @ z)
Cl₂ @ z  →  z
Cl₃(x) @ z  →  x::z
comp₁(f) @ g  →  Cl₁(f,g)
comp @ f  →  comp₁(f)
match_walk([])  →  Cl₂
match_walk(x::ys)  →
  comp @ (fix_walk @ ys) @ Cl₃(x)
walk @ xs  →  match_walk(xs)
fix_walk @ []  →  Cl₂
fix_walk @ (x::ys)  →
  Cl₁(fix_walk @ ys,Cl₃(x))
```

```
11  rev @ l   →  fix_walk @ l @ []
12  main(l)   →  fix_walk @ l @ []
```

The involved inlining rules are all non-erasing, i.e. all inlinings are *redex preserving*. As a consequence of Corollary 1, a bound on the runtime complexity of the above system can be relayed, within a constant multiplier, back to the ATRS $\mathcal{A}_{\mathsf{rev}}$.

Note that the modified system from Example 6 gives further possibilities for inlining. For instance, we could narrow further down the call to $\mathtt{fix}_{\mathtt{walk}}$ in rules (10), (11) and (12), performing case analysis on the variable *ys* and *l*, respectively. Proceeding this way would step-by-step unfold the definition of $\mathtt{fix}_{\mathtt{walk}}$, *ad infinitum*. We could have also further reduced the rules defining $\mathtt{match}_{\mathtt{walk}}$ and `walk`. However, it is not difficult to see that these rules will never be unfolded in a call to `main`, they have been sufficiently inlined and can be removed. Elimination of such *unused* rules will be discussed next.

4.2 Elimination of Dead Code

The notion of *usable rules* is well-established in the rewriting community. Although its precise definition depends on the context used (e.g. termination [4] and complexity analysis [30]), the notion commonly refers to a syntactic method for detecting that certain rules can never be applied in derivations starting from a given set of terms. From a programming point of view, such rules correspond to *dead code*, which can be safely eliminated.

Dead code arises frequently in automated program transformations, and its elimination turns out to facilitate our transformation-based approach to complexity analysis. The following definition formalises *dead code elimination* abstractly, for now. Call a rule $l \to r \in \mathcal{A}$ *usable* if it can be applied in a derivation

$$\mathtt{main}(d_1, \dots, d_k) \to_{\mathcal{A}} \cdots \to_{\mathcal{A}} t_1 \to_{\{l \to r\}} t_2 \ ,$$

where $d_i \in \mathsf{Input}$. The rule $l \to r$ is *dead code* if it is not usable. The following proposition follows by definition.

Proposition 4 (Dead Code Elimination)**.** *Dead code elimination, which maps an ATRS $\mathcal{A}$ to a subset of $\mathcal{A}$ by removing dead code only, is complexity reflecting and preserving.*

It is not computable in general which rules are dead code. One simple way to eliminate dead code is to collect all the function symbols underlying the definition of `main`, and remove the defining rules of symbols not in this collection, compare e.g. [30]. This approach works well for standard TRSs, but is usually inappropriate for ATRSs where most rules define a single function symbol, the application symbol. A conceptually similar, but unification based, approach that works reasonably well for ATRSs is given in [24]. However, the accurate identification of dead code, in particular in the presence of higher-order functions, requires more than just a simple syntactic analysis. We show in Section 5.2 a particular form of control flow analysis which leverages dead code elimination. The following example indicates that such an analysis is needed.

Example 7. We revisit the simplified ATRS from Example 6. The presence of the composition rule (1), itself a usable rule, makes it harder to infer which of the application rules are dead code. Indeed, the unification-based method found in [24] classifies all rules as usable. As we hinted in Section 2, the variables *f* and *g* are instantiated only by a very limited number of closures in a call of `main(`*l*`)`. In particular, none of the symbols `rev`, `walk`, `comp` and $\mathtt{comp}_1$ are passed to $\mathtt{Cl}_1$. With this knowledge, it is not difficult to see that their defining rules, together with the rules defining $\mathtt{match}_{\mathtt{walk}}$, can be eliminated by Proposition 4. Overall, the complexity of the ATRS depicted in Example 6 is thus reflected by the ATRS consisting of the following six rules.

```
1  Cl_1(f,g) @ x   →  f @ (g @ x)
2  Cl_2 @ z   →  z
3  Cl_3(x) @ z   →  x::z
4  fix_walk @ []   →  Cl_2
5  fix_walk @ (x::ys)  →
     Cl_1(fix_walk @ ys,Cl_3(x))
6  main(l)   →  fix_walk @ l @ []
```

4.3 Instantiation

Inlining and dead code elimination can indeed help in simplifying defunctionalised programs. There is however a feature of ATRS they cannot eliminate in general, namely rules whose right-hand-sides have *head variables*, i.e. variables that occur to the left of an application symbol and thus denote a function. The presence of such rules prevents FOPs to succeed in all but trivial cases. The ATRS from Example 7, for instance, still contains one such rule, namely rule (1), with head variables *f* and *g*. The main reason FOPs perform poorly on ATRSs containing such rules is that they lack a sufficiently powerful form of control flow analysis, and they are thus unable to realise that function symbols simulating higher-order combinators are passed arguments of a very specific shape, and are thus often harmless. This is the case, as an example, for the function symbol $\mathtt{Cl}_1$.

The way out consists in specialising the ATRS rules. This has the potential of highlighting the *absence* of certain dangerous patterns, but of course must be done with great care, without hiding complexity under the carpet of non-exhaustive instantiation. All this can be formalised as follows.

Call a rule $s \to t$ an *instance* of a rule $l \to r$, if there is a substitution σ with $s = l\sigma$ and $t = r\sigma$. We say that an ATRS $\mathcal{B}$ is an *instantiation* of $\mathcal{A}$ iff all rules in $\mathcal{B}$ are instances of rules from $\mathcal{A}$. This instantiation is *sufficiently exhaustive* if for every derivation

$$\mathtt{main}(d_1, \dots, d_k) \to_{\mathcal{A}} t_1 \to_{\mathcal{A}} t_2 \to_{\mathcal{A}} \cdots \ ,$$

where $d_i \in \mathsf{Input}$, there exists a corresponding derivation

$$\mathtt{main}(d_1, \dots, d_k) \to_{\mathcal{B}} t_1 \to_{\mathcal{B}} t_2 \to_{\mathcal{B}} \cdots \ .$$

The following theorem is immediate from the definition.

Theorem 2 (Instantiation Transformation)**.** *Every* instantiation transformation, *mapping any ATRS into a sufficiently exhaustive instantiation of it, is complexity reflecting and preserving.*

Example 8 (Continued from Example 7)**.** We instantiate the rule $\mathtt{Cl}_1(f,g)$ `@` $x \to f$ `@` $(g$ `@` $x)$ by the two rules

$$\mathtt{Cl}_1(\mathtt{Cl}_2,\mathtt{Cl}_3(x)) \ @ \ z \to \mathtt{Cl}_3(x) \ @ \ (\mathtt{Cl}_2 \ @ \ z)$$
$$\mathtt{Cl}_1(\mathtt{Cl}_1(f,g),\mathtt{Cl}_3(x)) \ @ \ z \to \mathtt{Cl}_1(f,g) \ @ \ (\mathtt{Cl}_2 \ @ \ z) \ ,$$

leaving all other rules from the TRS depicted in Example 7 intact. As we reasoned already before, the instantiation is sufficiently exhaustive: in a reduction of `main(`*l*`)` for a list *l*, arguments to $\mathtt{Cl}_1$ are always of the form as indicated in the two rules. Note that the right-hand side of both rules can be reduced by inlining the calls in the right argument. Overall, we conclude that the runtime complexity of our running example is reflected in the ATRS consisting of the following six rules:

```
1  Cl_2 @ z   →  z
2  Cl_1(Cl_2,Cl_3(x)) @ z   →  x::z
3  Cl_1(Cl_1(f,g),Cl_3(x)) @ z  →
     Cl_1(f,g) @ (x::z)
4  fix_walk @ []   →  Cl_2
```

$$\begin{array}{ll}
5 & \texttt{fix}_{\texttt{walk}}\ \texttt{@}\ (x\texttt{::}ys) \ \rightarrow \\
 & \quad \texttt{Cl}_1(\texttt{fix}_{\texttt{walk}}\ \texttt{@}\ ys,\texttt{Cl}_3(x)) \\
6 & \texttt{main}(l) \ \rightarrow \ \texttt{fix}_{\texttt{walk}}\ \texttt{@}\ l\ \texttt{@}\ \texttt{[]}
\end{array}$$

4.4 Uncurrying

The ATRS from Example 8 is now sufficiently instantiated: for all occurrences of the `@` symbol, we know *which function* we are applying, even if we do not necessarily know *to what* we are applying it. The ATRS is not yet ready to be processed by FOPs, simply because the application symbol is anyway still there, and cannot be dealt with.

At this stage, however, the ATRS can indeed be brought to a form suitable for analysis by FOPs through *uncurrying*, see e.g. the account of Hirokawa et al. [32]. Uncurrying an ATRS $\mathcal{A}$ involves the definition of a fresh function symbol $\texttt{f}^n$ for each n-ary application

$$\texttt{f}(t_1,\dots,t_m)\ \texttt{@}\ t_{m+1}\ \texttt{@}\cdots\texttt{@}\ t_{m+n}\ ,$$

encountered in $\mathcal{A}$. This way, applications can be completely eliminated. Although in [32] only ATRSs defining function symbols of arity zero are considered, the extension to our setting poses no problem. We quickly recap the central definitions.

Define the *applicative arity* $\texttt{aa}_{\mathcal{A}}(\texttt{f})$ of a symbol $\texttt{f}$ in $\mathcal{A}$ as the maximal $n \in \mathbb{N}$ such that a term

$$\texttt{f}(t_1,\dots,t_m)\ \texttt{@}\ t_{m+1}\ \texttt{@}\cdots\texttt{@}\ t_{m+n}\ ,$$

occurs in $\mathcal{A}$.

Definition 1. The *uncurrying* $\llcorner t \lrcorner$ of a term $t = \texttt{f}(t_1,\dots,t_m)\ \texttt{@}\ t_{m+1}\ \texttt{@}\cdots\texttt{@}\ t_{m+n}$, with $n \leqslant \texttt{aa}_{\mathcal{A}}(\texttt{f})$ is defined as

$$\llcorner t \lrcorner := \texttt{f}^n(\llcorner t_1 \lrcorner,\dots,\llcorner t_m \lrcorner,\llcorner t_{m+1} \lrcorner,\dots,\llcorner t_{m+n} \lrcorner)\ ,$$

where $\texttt{f}^0 = \texttt{f}$ and $\texttt{f}^n$ $(1 \le n \le \texttt{aa}_{\mathcal{A}}(\texttt{f}))$ are fresh function symbols. Uncurrying is homomorphically extended to ATRSs.

Note that $\llcorner \mathcal{A} \lrcorner$ is well-defined whenever $\mathcal{A}$ is *head variable free*, i.e. does not contain a term of the form $x\ \texttt{@}\ t$ for variable x. We intend to use the TRS $\llcorner \mathcal{A} \lrcorner$ to simulate reductions of the ATRS $\mathcal{A}$. In the presence of rules of functional type however, such a simulation fails. To overcome the issue, we *η-saturate* $\mathcal{A}$.

Definition 2. We call an ATRS $\mathcal{A}$ *η-saturated* if whenever

$$\texttt{f}(l_1,\dots,l_m)\ \texttt{@}\ l_{m+1}\ \texttt{@}\cdots\texttt{@}\ l_{m+n} \to r \in \mathcal{A} \text{ with } n < \texttt{aa}_{\mathcal{A}}(\texttt{f}),$$

then it contains also a rule

$$\texttt{f}(l_1,\dots,l_m)\ \texttt{@}\ l_{m+1}\ \texttt{@}\cdots\texttt{@}\ l_{m+n}\ \texttt{@}\ z \to r\ \texttt{@}\ z\ ,$$

where z is a fresh variable. The *η-saturation* $\mathcal{A}_\eta$ of $\mathcal{A}$ is defined as the least extension of $\mathcal{A}$ that is η-saturated.

Remark. The η-saturation $\mathcal{A}_\eta$ of an ATRS $\mathcal{A}$ is not necessarily finite. A simple example is the one-rule ATRS $\texttt{f} \to \texttt{f}\ \texttt{@}\ \texttt{a}$ where both $\texttt{f}$ and $\texttt{a}$ are function symbols. Provided that the ATRS $\mathcal{A}$ is endowed with simple types, and indeed the simple typing of our initial program is preserved throughout our complete transformation pipeline, the η-saturation of $\mathcal{A}$ becomes finite.

Example 9 (Continued from Example 8)**.** The ATRS from Example 8 is not η-saturated: $\texttt{fix}_{\texttt{walk}}$ is applied to two arguments in rule (6), but its defining rules, rule (4) and (5), take a single argument only. The η-saturation thus contains in addition the following two rules

$$\texttt{fix}_{\texttt{walk}}\ \texttt{@}\ \texttt{[]}\ \texttt{@}\ z \to \texttt{Cl}_2\ \texttt{@}\ z\ ;$$
$$\texttt{fix}_{\texttt{walk}}\ \texttt{@}\ (x\texttt{::}ys)\ \texttt{@}\ z \to \texttt{Cl}_1(\texttt{fix}_{\texttt{walk}}\ \texttt{@}\ ys,\texttt{Cl}_3(x))\ \texttt{@}\ z\ .$$

One can then check that the resulting system is η-saturated.

Lemma 2. *Let $\mathcal{A}_\eta$ be the η-saturation of $\mathcal{A}$.*

1. *The rewrite relation $\to_{\mathcal{A}_\eta}$ coincides with $\to_{\mathcal{A}}$.*
2. *Suppose $\mathcal{A}_\eta$ is head variable free. If $s \to_{\mathcal{A}_\eta} t$ then $\llcorner s \lrcorner \to_{\llcorner \mathcal{A}_\eta \lrcorner} \llcorner t \lrcorner$.*

Proof. For Property 1, the inclusion $\to_{\mathcal{A}} \subseteq \to_{\mathcal{A}_\eta}$ follows trivially from the inclusion $\mathcal{A} \subseteq \mathcal{A}_\eta$. The inverse inclusion $\to_{\mathcal{A}} \supseteq \to_{\mathcal{A}_\eta}$ can be shown by a standard induction on the derivation of $l \to r \in \mathcal{A}_\eta$.

Property 2 can be proven by induction on t. The proof follows the pattern of the proof of Sternagel and Thiemann [51]. Notice that in [51, Theorem 10], the rewrite system $\llcorner \mathcal{A}_\eta \lrcorner$ is enriched with uncurrying rules of the form $\texttt{f}^i(x_1,\dots,x_n)\ \texttt{@}\ y \to \texttt{f}^{i+1}(x_1,\dots,x_n,y)$. Such an extension is not necessary in the absence of head variables. In our setting, the application symbol is completely eliminated by uncurrying, and thus the above rules are dead code. □

As a consequence, we immediately obtain the following theorem.

Theorem 3 (Uncurrying Transformation)**.** *Suppose that $\mathcal{A}$ is head variable free. The* uncurrying *transformation, which maps an ATRS $\mathcal{A}$ to the system $\llcorner \mathcal{A}_\eta \lrcorner$, is complexity reflecting.*

Example 10 (Continued from Example 9)**.** Uncurrying the η-saturated ATRS, consisting of the six rules from Example 8 and the two rules from Example 9, results in the following set of rules:

$$\begin{array}{l}
\texttt{Cl}_1^1(\texttt{Cl}_2,\texttt{Cl}_3(x),z) \ \rightarrow \ x\texttt{::}z \\
\texttt{Cl}_1^1(\texttt{Cl}_1(f,g),\texttt{Cl}_3(x),z) \ \rightarrow \ \texttt{Cl}_1^1(f,g,x\texttt{::}z) \\
\texttt{Cl}_2^1(z) \ \rightarrow \ z \\
\texttt{fix}_{\texttt{walk}}^1(\texttt{[]}) \ \rightarrow \ \texttt{Cl}_2 \\
\texttt{fix}_{\texttt{walk}}^1(x\texttt{:}ys) \ \rightarrow \ \texttt{Cl}_1(\texttt{fix}_{\texttt{walk}}^1(ys),\texttt{Cl}_3(x)) \\
\texttt{fix}_{\texttt{walk}}^2(\texttt{[]},z) \ \rightarrow \ \texttt{Cl}_2^1(z) \\
\texttt{fix}_{\texttt{walk}}^2(x\texttt{::}ys,z) \ \rightarrow \\
\quad \texttt{Cl}_{11}(\texttt{fix}_{\texttt{walk}}^1(ys),\texttt{Cl}_3(x),z) \\
\texttt{main}(l) \ \rightarrow \ \texttt{fix}_{\texttt{walk}}^2(l,\texttt{[]})
\end{array}$$

Inlining the calls to $\texttt{fix}_{\texttt{walk}}^2$ and $\texttt{Cl}_2^1(z)$, followed by dead code elimination, results finally in the TRS $\mathcal{R}_{\textsf{rev}}$ from Section 2.

5. Automation

In the last section we have laid the formal foundation of our program transformation methodology, and ultimately of our tool HOCA. Up to now, however, program transformations (except for uncurrying) are too abstract to be turned into actual algorithms. In dead code elimination, for instance, the underlying computation problem (namely the one of precisely isolating usable rules) is undecidable. In inlining, one has a decidable transformation, which however results in a blowup of program sizes, if blindly applied.

This section is devoted to describing some *concrete* design choices we made when automating our program transformations. Another, related, issue we will talk about is the effective *combination* of these techniques, the *transformation pipeline*.

5.1 Automating Inlining

The main complication that arises while automating our inlining transformation is to decide *where* the transformation should be applied. Here, there are two major points to consider: first, we want to ensure that the overall transformation is not only complexity *reflecting*, but also complexity *preserving*, thus not defeating its purpose. To address this issue, we employ inlining conservatively, ensuring that it does not duplicate function calls. Secondly, as we already hinted after Example 6, exhaustive inlining is usually not desirable and may even lead to non-termination in the transformation pipeline described below. Instead, we want to ensure that inlining *simplifies* the problem with respect to some sensible *metric*, and plays well in conjunction with the other transformation techniques.

Instead of working with a closed inlining strategy, our implementation `inline(`P`)` is parameterised by a predicate P which, intuitively, tells when inlining a call at position p in a rule $l \to r$ is *sensible* at the current stage in our transformation pipeline. The algorithm `inline(`P`)` replaces every rule $l \to r$ by $\mathsf{inline}_{\mathcal{A},p}(l \to r)$ for some position p such that $P(p, l \to r)$ holds. The following four predicates turned out to be useful in our transformation pipeline. The first two are designed by taking into account the specific shape of ATRSs obtained by defunctionalisation, the last two are generic.

- `match`: This predicate holds if the right-hand side r is labeled at position p by a symbol of the form $\mathtt{match}_{cs}$. That is, the predicate enables inlining of calls resulting from the translation of a match-expression, thereby eliminating one indirection due to the encoding of pattern matching during defunctionalisation.
- `lambda-rewrite`: This predicate holds if the subterm $r|_p$ is of the form $\mathtt{lam}_{x.e}(\vec{t})$ `@` s. By definition it is enforced that inlining corresponds to a plain rewrite, head variables are not instantiated. For instance, `inline(lambda-rewrite)` is inapplicable on the rule $\mathtt{Cl}_2(f,g)$ `@` $z \to f$ `@` `(`g `@` z`)`. This way, we avoid that variables f and g are improperly instantiated.
- `constructor`: The predicate holds if the right-hand sides of all rules used to inline $r|_p$ are constructor terms, i.e. do not give rise to further function calls. Overall, the number of function calls therefore decreases. As a side effect, more patterns become obvious in rules, which facilitates further inlining.
- `decreasing`: The predicate holds if any of the following two conditions is satisfied: (i) *proper inlining*: the subterm $r|_p$ constitutes the only call-site to the inlined function `f`. This way, all rules defining `f` in $\mathcal{A}$ will turn to dead code after inlining. (ii) *size decreasing*: each right-hand side in $\mathsf{inline}_{\mathcal{A},p}(l \to r)$ is strictly smaller in size than the right-hand side r. The aim is to facilitate FOPs on the generated output. In the first case, the number of rules decreases, which usually implies that in the analysis, a FOP generates less constraints which have to be solved. In the second case, the number of constraints might increase, but the individual constraints are usually easier to solve.

We emphasise that all inlinings performed on our running example $\mathcal{A}_{\mathsf{rev}}$ are captured by the instances of inlining just defined.

5.2 Automating Instantiation and Dead Code Elimination via Control Flow Analysis

One way to effectively eliminate dead code and apply instantiation, as in Examples 7 and 8, consists in inferring the shape of closures passed during reductions. This way, we can on the one hand specialise rewrite rules being sure that the obtained instantiation is sufficiently exhaustive, and on the other hand discover that certain rules are simply useless, and can thus be eliminated.

To this end, we rely on an approximation of the collecting semantics. In static analysis, the collecting semantics of a program maps a given program point to the collection of states attainable when control reaches that point during execution. In the context of rewrite systems, it is natural to define the rewrite rules as program points, and substitutions as states. Throughout the following, we fix an ATRS $\mathcal{A} = \{l_i \to r_i\}_{i \in \{1,\dots,n\}}$. We define the *collecting semantics of* $\mathcal{A}$ as a tuple $(Z_1, \dots, Z_n)$, where

$$\begin{aligned} Z_i := \{(\sigma, t) \mid \exists \vec{d} \in \mathsf{Input}.\\ \mathtt{main}(\vec{d}) \to_{\mathcal{A}}^{*} C[l_i\sigma] \to_{\mathcal{A}} C[r_i\sigma] \text{ and } r_i\sigma \to_{\mathcal{A}}^{*} t\}\ . \end{aligned}$$

Here the substitutions σ are restricted to the set $\mathsf{Var}(l_i)$ of variables occurring in the left-hand side in l_i.

The collecting semantics of $\mathcal{A}$ includes all the necessary information for implementing both dead code elimination and instantiation:

Lemma 3. *The following properties hold:*

1. *The rule $l_i \to r_i \in \mathcal{A}$ constitutes dead code if and only if $Z_i = \varnothing$.*
2. *Suppose the ATRS $\mathcal{B}$ is obtained by instantiating rules $l_i \to r_i$ with substitutions $\sigma_{i,1}, \dots, \sigma_{i,k_i}$. Then the instantiation is sufficiently exhaustive if for every substitution σ with $(\sigma, t) \in Z_i$, there exists a substitution $\sigma_{i,j}$ $(j \in \{1, \dots, i_k\})$ which is at least as general as σ.*

Proof. The first property follows by definition. For the second property, consider a derivation

$$\mathtt{main}(d_1, \dots, d_k) \to_{\mathcal{A}}^{*} C[l_i\sigma] \to_{\mathcal{A}} C[r_i\sigma]\ ,$$

and thus $(\sigma, r_i\sigma) \in Z_i$. By assumption, there exists a substitution $\sigma_{i,j}$ $(i \in \{1, \dots, i_k\})$ is at least as general as σ. Hence the ATRS $\mathcal{B}$ can simulate the step from $C[l_i\sigma] \to_{\mathcal{A}} C[r_i\sigma]$, using the rule $l_i\sigma_{i,j} \to r_i\sigma_{i,j} \in \mathcal{B}$. From this, the property follows by inductive reasoning. □

As a consequence, the collecting semantics of $\mathcal{A}$ is itself not computable. Various techniques to over-approximate the collecting semantics have been proposed, e.g. by Feuillade et al. [23], Jones [33] and Kochems and Ong [36]. In all these works, the approximation consists in describing the tuple $(Z_1, \dots, Z_n)$ by a *finite* object.

In `HOCA` we have implemented a variation of the technique of Jones [33], tailored to call-by-value semantics (already hinted at in [33]). Conceptually, the form of control flow analysis we perform is close to a 0-CFA [41], merging information derived from different call sites. Whilst being efficient to compute, the precision of this relatively simple approximation turned out to be reasonable for our purpose.

The underlying idea is to construct a *(regular) tree grammar* which over-approximates the collecting semantics. Here, a *tree grammar* $\mathcal{G}$ can be seen as a ground ATRS whose left-hand sides are all function symbols with arity zero. The *non-terminals* of $\mathcal{G}$ are precisely the left-hand sides. For the remaining, we assume that variables occurring $\mathcal{A}$ are indexed by indices of rules, i.e. every variable occurring in the ith rule $l_i \to r_i \in \mathcal{A}$ has index i. Hence the set of variables of rewrite rules in $\mathcal{A}$ are pairwise disjoint. Besides a designated non-terminal S, the *start-symbol*, the constructed tree grammar $\mathcal{G}$ admits two kinds of non-terminals: non-terminals R_i for each rule $l_i \to r_i$ and non-terminals z_i for variables z_i occurring in $\mathcal{A}$. Note that the variable z_i is considered as a constant in $\mathcal{G}$. We say that $\mathcal{G}$ is *safe* for $\mathcal{A}$ if the following two conditions are satisfied for all $(\sigma, t) \in Z_i$: (i) $z_i \to_{\mathcal{G}}^{*} \sigma(z_i)$ for each $z_i \in \mathsf{Var}(l_i)$; and (ii) $R_i \to_{\mathcal{G}}^{*} t$. This way, $\mathcal{G}$ constitutes a finite over-approximation of the collecting semantics of $\mathcal{A}$.

Example 11. Figure 2 shows the tree grammar $\mathcal{G}$ constructed by the method described below, which is safe for the ATRS $\mathcal{A}$ from Example 6. The notation $N \to t_1 \mid \cdots \mid t_n$ is a short-hand for the n rules $N \to t_i$.

The construction of Jones consists of an initial automaton $\mathcal{G}_0$, which describes considered start terms, and which is then systematically closed under rewriting by way of an extension operator $\delta(\cdot)$. Suitable to our concerns, we define $\mathcal{G}_0$ as the tree grammar consisting of the following rules:

$$\begin{aligned} &S \to \mathtt{main}(*, \dots, *) && \text{and} \\ &* \to \mathtt{C}_j(*, \dots, *) && \text{for each constructor } \mathtt{C}_j \text{ of } \mathcal{A}. \end{aligned}$$

$$\begin{array}{rlrlrl}
\ast & \to \texttt{[]} \mid \ast\texttt{::}\ast & R_1 & \to f_1 \texttt{@} \texttt{(} g_1 \texttt{@}\ z_1 \texttt{)} \mid f_1 \texttt{@} R_3 \mid R_1 \mid R_2 & z_3 & \to z_1 \\
S & \to \texttt{main(}\ast\texttt{)} \mid R_{12} & R_3 & \to x_3\texttt{::}z_3 & f_1 & \to R_9 \mid R_{10} \\
R_{12} & \to \texttt{fix}_{\texttt{walk}} \texttt{@}\ l_{12} \texttt{@ []} \mid R_9 \texttt{@ []} \mid R_{10} \texttt{@ []} \mid R_1 \mid R_2 & R_2 & \to z_2 & g_1 & \to \texttt{Cl}_3\texttt{(}x_{10}\texttt{)} \\
 & \mid \texttt{Cl}_1\texttt{(}R_9\texttt{,Cl}_3\texttt{(}x_{10}\texttt{))} \mid \texttt{Cl}_1\texttt{(}R_{10}\texttt{,Cl}_3\texttt{(}x_{10}\texttt{))} & l_{12} & \to \ast & z_1 & \to R_3 \mid \texttt{[]} \\
R_{10} & \to \texttt{Cl}_1\texttt{(fix}_{\texttt{walk}}\texttt{@}\ ys_{10}\texttt{,Cl}_3\texttt{(}x_{10}\texttt{))} & x_{10} & \to \ast & z_2 & \to R_3 \mid \texttt{[]} \\
R_9 & \to \texttt{Cl}_2 & ys_{10} & \to \ast & x_3 & \to x_{10}
\end{array}$$

Figure 2. Over-approximation of the collecting semantics of the ATRS from Example 6.

Then clearly $S \to_{\mathcal{G}}^* \texttt{main}(d_1, \ldots, d_n)$ for all inputs $d_i \in \mathsf{Input}$. We let $\mathcal{G}$ be the least set of rules satisfying $\mathcal{G} \supseteq \mathcal{G}_0 \cup \delta(\mathcal{G})$ with

$$\delta(\mathcal{G}) := \bigcup_{N \to C[u] \in \mathcal{G}} \mathsf{Ext}^{\mathsf{cbv}}(N \to C[u]) \ .$$

Here, $\mathsf{Ext}^{\mathsf{cbv}}(N \to C[u])$ is defined as the following set of rules:

$$\left\{ \begin{array}{l|l} N \to C[R_i], & l_i \to r_i \in \mathcal{A}, \\ R_i \to r_i, \text{ and} & u \to_{\mathcal{G}}^* l_i\sigma \text{ is minimal} \\ z_i \to \sigma(z_i) \text{ for all } z_i \in \mathsf{Var}(l_i) & \text{and } \sigma \text{ normalised wrt. } \mathcal{A}. \end{array} \right\}$$

In contrast to [33], we require that the substitution σ is normalised, thereby modelling call-by-value semantics. The tree grammar $\mathcal{G}$ is computable using a simple fix-point construction. Minimality of $\texttt{f}(t_1, \ldots, t_k) \to_{\mathcal{G}}^* l_i\sigma$ means that there is no shorter sequence $\texttt{f}(t_1, \ldots, t_k) \to_{\mathcal{G}}^* l_i\tau$ with $l_i\tau \to_{\mathcal{G}}^* l_i\sigma$, and ensures that $\mathcal{G}$ is finite [33], thus the construction is always terminating.

We illustrate the construction on the ATRS from Example 6.

Example 12. Revise the ATRS from Example 6. To construct the safe tree grammar as explained above, we start from the initial grammar $\mathcal{G}_0$ given by the rule

$$S \to \texttt{main(}\ast\texttt{)} \qquad\qquad \ast \to \texttt{[]} \mid \ast\texttt{::}\ast \ ,$$

and then successively fix violations of the above closure condition. The only violation in the initial grammar is caused by the first production. Here, the right-hand side $\texttt{main(}\ast\texttt{)}$ matches the (renamed) rule 12: $\texttt{main(}l_{12}\texttt{)} \to \texttt{fix}_{\texttt{walk}}\texttt{@}\ l_{12}\ \texttt{@ []}$, using the substitution $\{l_{12} \mapsto \ast\}$. We fix the violation by adding productions

$$S \to R_{12} \qquad R_{12} \to \texttt{fix}_{\texttt{walk}} \texttt{@}\ l_{12} \texttt{@ []} \qquad l_{12} \to \ast \ .$$

The tree grammar $\mathcal{G}$ constructed so far tells us that l_{12} is a list. In particular, we have the following two minimal sequences which makes the left subterm of the R_{12}-production an instances of the left-hand sides of defining rules of $\texttt{fix}_{\texttt{walk}}$ (rules (9) and (10)):

$$\begin{aligned} \texttt{fix}_{\texttt{walk}} \texttt{@}\ l_{12} &\to_{\mathcal{G}}^+ \texttt{fix}_{\texttt{walk}} \texttt{@ []} \ , \\ \texttt{fix}_{\texttt{walk}} \texttt{@}\ l_{12} &\to_{\mathcal{G}}^+ \texttt{fix}_{\texttt{walk}} \texttt{@}\ \ast\texttt{::}\ast \ . \end{aligned}$$

To resolve the closure violations, the tree grammar is extended by productions

$$R_{12} \to R_9 \texttt{@ []} \qquad\qquad R_9 \to \texttt{Cl}_2$$

because of rule (9), and by

$$\begin{array}{ll} R_{12} \to R_{10} \texttt{@ []} & x_{10} \to \ast \\ R_{10} \to \texttt{Cl}_1\texttt{(fix}_{\texttt{walk}}\texttt{@}\ ys_{10}\texttt{,Cl}_3\texttt{(}x_{10}\texttt{))} & ys_{10} \to \ast \ . \end{array}$$

due to rule (10). We can now identify a new violation in the production of R_{10}. Fixing all violations this way will finally result in the tree grammar depicted in Figure 2.

The following lemma confirms that $\mathcal{G}$ is closed under rewriting with respect to the call-by-value semantics. The lemma constitutes a variation of Lemma 5.3 from [33].

Lemma 4. *If $S \to_{\mathcal{G}}^* t$ and $t \to_{\mathcal{A}}^* C[l_i\sigma] \to_{\mathcal{A}} C[r_i\sigma]$ then $S \to_{\mathcal{G}}^* C[R_i]$, $R_i \to_{\mathcal{G}} r_i$ and $z_i \to_{\mathcal{G}}^* \sigma(z_i)$ for all $z_i \in \mathsf{Var}(l_i)$.*

Theorem 4. *The tree grammar $\mathcal{G}$ is safe for $\mathcal{A}$.*

Proof. Fix $(\sigma, t) \in Z_i$, and let $z \in \mathsf{Var}(l_i)$. Thus $\texttt{main}(\vec{d}) \to_{\mathcal{A}}^* C[l_i\sigma] \to_{\mathcal{A}} C[r_i\sigma]$ and $r_i\sigma \to_{\mathcal{A}}^* t$ for some inputs $d \in \mathsf{Input}$. As we have $S \to_{\mathcal{G}}^* \texttt{main}(\vec{d})$ since $\mathcal{G}_0 \subseteq \mathcal{G}$, Lemma 4 yields $R_i \to_{\mathcal{G}} r_i$ and $z_i \to_{\mathcal{G}}^* \sigma(z)$, i.e. the second safeness conditions is satisfied. Clearly, $R_i \to_{\mathcal{G}} r_i \to_{\mathcal{G}}^* r_i\sigma$. A standard induction on the length of $r_i\sigma \to_{\mathcal{A}}^* t$ then yields $R_i \to_{\mathcal{A}}^* t$, using again Lemma 4. □

We arrive now at our concrete implementation $\texttt{cfa}(\mathcal{A})$ that employs the above outlined call flow analysis to deal with both dead code elimination and instantiation on the given ATRS $\mathcal{A}$. The construction of the tree grammar $\mathcal{G}$ follows itself closely the algorithm outlined by Jones [33]. Recall that the i^{th} rule $l_i \to r_i \in \mathcal{A}$ constitutes dead code if the i^{th} component Z_i of the collecting semantics of $\mathcal{A}$ is empty, by Lemma 3(1). Based on the constructed tree grammar, the implementation identifies rule $l_i \to r_i$ as dead code when $\mathcal{G}$ does not define a production $R_i \to t$ and thus $Z_i = \varnothing$. All such rules are eliminated, in accordance to Proposition 4. On the remaining rules, our implementation performs instantiation as follows. We suppose *ϵ-productions* $N \to M$, for non-terminals M, have been eliminated by way of a standard construction, preserving the set of terms from non-terminals in $\mathcal{G}$. Thus productions in $\mathcal{G}$ have the form $N \to \texttt{f}(t_1, \ldots, t_k)$. Fix a rule $l_i \to r_i \in \mathcal{A}$. The primary goal of this stage is to get rid of head variables, with respect to the η-saturated ATRS $\mathcal{A}_\eta$, thereby enabling uncurrying so that the ATRS $\mathcal{A}$ can be brought into functional form. For all such head variables z, then, we construct a set of binders

$$\{z_i \mapsto \mathsf{fresh}(\texttt{f}(t_1, \ldots, t_k)) \mid z_i \to \texttt{f}(t_1, \ldots, t_k) \in \mathcal{G}\} \ ,$$

where the function fresh replaces non-terminals by fresh variables, discarding binders where the right-hand contains defined symbols. For variables z which do not occur in head positions, we construct such a binder only if the production $z_i \to \texttt{f}(t_1, \ldots, t_k)$ is unique. With respect to the tree grammar of Figure 2, the implementation generates binders

$$\{f_1 \mapsto \texttt{Cl}_2, f_1 \mapsto \texttt{Cl}_1\texttt{(}f\texttt{'}\texttt{,Cl}_3\texttt{(}x\texttt{'))}\} \text{ and } \{g_1 \mapsto \texttt{Cl}_3\texttt{(}x\texttt{'')}\} \ .$$

The product-combination of all such binders gives then a set of substitution $\{\sigma_{i,1}, \ldots, \sigma_{i,i_k}\}$ that leads to sufficiently many instantiations $l_i\sigma_{i,j} \to r_i\sigma_{i,j}$ of rule $l_i \to r_i$, by Lemma 3(2). Our implementation replaces every rule $l_i \to r_i \in \mathcal{A}$ by instantiations constructed this way.

The definition of binder was chosen to keep the number of computed substitutions minimal, and hence the generated head variable free ATRS small. Putting things together, we see that the instantiation is sufficiently exhaustive, and thus the overall transformation is complexity reflecting and preserving by Theorem 2. By $\texttt{cfaDCE}$ we denote the variation of $\texttt{cfa}$ that performs dead code elimination, but no instantiations.

```
simplify = simpATRS; toTRS; simpTRS where
  simpATRS =
    exhaustive inline(lambda-rewrite);
    exhaustive inline(match);
    exhaustive inline(constructor);
    usableRules
  toTRS = cfa; uncurry; usableRules
  simpTRS =
    exhaustive ((inline(decreasing);
                 usableRules) > cfaDCE)
```

Figure 3. Transformation Strategy in HOCA.

5.3 Combining Transformations

We have now seen all the building blocks underlying our tool HOCA. But *in which order* should we apply the introduced program transformations? In principle, one could try to blindly iterate the proposed techniques and hope that a FOP can cope with the output. Since transformations are closed under composition, the blind iteration of transformations is sound, although seldom effective. In short, a *strategy* is required that combines the proposed techniques in a sensible way. There is no clear notion of a perfect strategy. After all, we are interested in non-trivial program properties. However, it is clear that any sensible strategy should at least (i) yield overall a transformation that is effectively computable, (ii) generate ATRSs whose runtime complexity is in relation to the complexity of the analysed program, and (iii) produce ATRSs suitable for analysis via FOPs.

In Figure 3 we render the transformation strategy underlying our tool HOCA. More precise, Figure 3 defines a transformation `simplify` based on the following *transformation combinators*:

- $f_1\,;f_2$ denotes the composition $f_2 \circ \underline{f_1}$, where $\underline{f_1}(\mathcal{A}) = f_1(\mathcal{A})$ if defined and $\underline{f_1}(\mathcal{A}) = \mathcal{A}$ otherwise;
- the transformation **exhaustive** f iterates the transformation f until inapplicable on the current ATRS; and
- the operator > implements left-biased choice: $f_1 \;>\; f_2$ applies transformation f_1 if successful, otherwise f_2 is applied.

It is easy to see that all three combinators preserve the two crucial properties of transformations, viz, complexity reflection and complexity preservation.

The transformation `simplify` depicted in Figure 3 is composed out of three transformations `simpATRS`, `toTRS` and `simpTRS`, each itself defined from transformations `inline`(P) and `cfa` describe in Sections 5.1 and 5.2, respectively, the transformation `usableRules` which implements the aforementioned computationally cheap, unification based, criterion from [24] to eliminate dead code (see Section 4.2), and the transformation `uncurry`, which implements the uncurrying-transformation from Section 4.4.

The first transformation in our chain, `simpATRS`, performs inlining driven by the specific shape of the input ATRS obtained by defunctionalisation, followed by syntax driven dead code elimination. The transformation `toTRS` will then translate the intermediate ATRSs to functional form by the uncurrying transformation, using control flow analysis to instantiate head variables sufficiently and further eliminate dead code. The transformation `simpTRS` then simplifies the obtained TRS by controlled inlining, applying syntax driven dead code elimination where possible, resorting to the more expensive version based on control flow analysis in case the simplification stales. To understand the sequencing of transformations in `simpTRS`, observe that the strategy `inline(decreasing)` is interleaved with dead code elimination. Dead code elimination, both in the form of `usableRules` and `cfaDCE`, potentially restricts the set $\mathsf{inline}_{\mathcal{A},p}(l \to r)$, and might facilitate in consequence the transformation `inline(decreasing)`. Importantly, the rather expensive, flow analysis driven, dead code analysis is only performed in case both `inline(decreasing)` and its cheaper cousin `usableRules` fail.

The overall strategy `simplify` is well-defined on all inputs obtained by defunctionalisation, i.e. terminating [10]. Although we cannot give precise bounds on the runtime complexity in general, in practice the number of applications of inlinings is sufficiently controlled to be of practical relevance. Importantly, the way inlining and instantiation is employed ensures that the sizes of all intermediate TRSs are kept under tight control.

6. Experimental Evaluation

So far, we have covered the theoretical and implementation aspects underlying our tool HOCA. The purpose of this section is to indicate how our methods performs in practice. To this end, we compiled a diverse collection of higher-order programs from the literature [22, 35, 43] and standard textbooks [15, 47], on which we performed tests with our tool in conjunction with the general-purpose first-order complexity tool TcT [8], version 2.1.[6] For comparison, we have also paired HOCA with the termination tool TTT2 [37], version 1.15.

In Table 1 we summarise our experimental findings on the 25 examples from our collection.[7] Row S in the table indicates the total number of higher-order programs whose runtime could be classified linear, quadratic and at most polynomial when HOCA is paired with the back-end TcT, and those programs that can be shown terminating when HOCA is paired with TTT2. In contrast, row D shows the same statistics when the FOP is run directly on the defunctionalised program, given by Proposition 2. To each of those results, we state the minimum, average and maximum execution time of HOCA and the employed FOP. All experiments were conducted on a machine with a 8 dual core AMD Opteron™ 885 processors running at 2.60GHz, and 64Gb of RAM.[8] Furthermore, the tools were advised to search for a certificate within 60 seconds.

As the table indicates, not all examples in the testbed are subject to a runtime complexity analysis through the here proposed approach. However, at least termination can be automatically verified. For all but one example (namely `mapplus.fp`) the obtained complexity certificate is asymptotically optimal. As far as we know, no other fully automatic complexity tool can handle the five open examples. We will comment below on the reason why HOCA may fail.

Let us now analyse some of the programs from our testbed. For each program, we will briefly discuss what HOCA, followed by selected FOPs can prove about it. This will give us the opportunity to discuss about specific aspects of our methodology, but also about limitations of the current FOPs.

Reversing a List. Our running example, namely the functional program from Section 2 which reverses a list, can be transformed by HOCA into a TRS which can easily be proved to have linear complexity. Similar results can be proved for other programs.

Parametric Insertion Sort. A more complicated example is a higher-order formulation of the insertion sort algorithm, example `isort-fold.fp`, which is parametric on the subroutine which compares the elements of the list being sorted. This is an example

[6] We ran also experiments with AProVE and CaT as back-end, this however did not extend the power.

[7] Examples and full experimental evidence can be found on the HOCA homepage.

[8] Average PassMark CPU Mark 2851; http://www.cpubenchmark.net/.

Table 1. Experimental Evaluation conducted with TCT and TTT$_2$.

		constant	linear	quadratic	polynomial	terminating
D	# systems	2	5	5	5	8
	FOP execution time	0.37 / 1.71 / 3.05	0.37 / 4.82 / 13.85	0.37 / 4.82 / 13.85	0.37 / 4.82 / 13.85	0.83 / 1.38 / 1.87
S	# systems	2	14	18	20	25
	`HOCA` execution time	0.01 / 2.28 / 4.56	0.01 / 0.54 / 4.56	0.01 / 0.43 / 4.56	0.01 / 0.42 / 4.56	0.01 / 0.87 / 6.48
	FOP execution time	0.23 / 0.51 / 0.79	0.23 / 2.53 / 14.00	0.23 / 6.30 / 30.12	0.23 / 10.94 / 60.10	0.72 / 1.43 / 3.43

which cannot be handled by linear type systems [13]: we do recursion over a function which in an higher-order variable occurs free. Also, type systems like the ones in [35], which are restricted to linear complexity certificates, cannot bind the runtime complexity of this program. `HOCA`, instead, is able to put it in a form which allows TCT to conclude that the complexity is, indeed quadratic.

Divide and Conquer Combinators. Another noticeable example is the divide an conquer combinator, defined in example `mergesort-dc.fp`, which we have taken from [47]. We have then instantiated it so that the resulting algorithm is the merge sort algorithm. `HOCA` is indeed able to translate the program into a first-order program which can then be proved to be *terminating* by FOPs. This already tells us that the obtained ATRS is in a form suitable for the analysis. The fact that FOPs cannot say anything about its complexity is due to the limitations of current FOPS which, indeed, are currently not able to perform a sufficiently powerful non-local size analysis, a necessary condition for proving merge sort to be a polynomial time algorithm. Similar considerations hold for Okasaki's parser combinator, various instances of which can be proved themselves terminating.

7. Related Work

What this paper shows is that complexity analysis of higher-order functional programs can be made easier by way of program transformations. As such, it can be seen as a complement rather than an alternative to existing methodologies. Since the literature on related work is quite vast, we will only give in this section an overview of the state of the art, highlighting the differences with to our work.

Control Flow Analysis. A clear understanding of control flow in higher-order programs is crucial in almost any analysis of non-functional properties. Consequently, the body of literature on control flow analysis is considerable, see e.g. the recent survey of Midtgaard [39]. Closest to our work, control flow analysis has been successfully employed in termination analysis, for brevity we mention only [25, 34, 44]. By Jones and Bohr [34] a strict, higher-order language is studied, and control flow analysis facilitates the construction of size-change graphs needed in the analysis. Based on earlier work by Panitz and Schmidt-Schauß [44], Giesl et al. [25] study termination of `Haskell` through so-called *termination* or *symbolic execution* graphs, which under the hood corresponds to a careful study of the control flow in `Haskell` programs. While arguable *weak dependency pairs* [30] or *dependency triples* [42] form a weak notion of control flow analysis, our addition of collecting semantics to complexity analysis is novel.

Type Systems. That the rôle of type systems can go beyond type safety is well-known. The abstraction type systems implicitly provide, can enforces properties like termination or bounded complexity. In particular, type systems for the λ-calculus are known which characterise relatively small classes of functions like the one of polynomial time computable functions [13]. The principles underlying these type systems, which by themselves cannot be taken as verification methodologies, have been leveraged while defining type systems for more concrete programming languages and type inference procedures, some of them being intensionally complete [18, 20]. All these results are of course very similar in spirit to what we propose in this work. What is lacking in most of the proposed approaches is the presence, at the same time, of higher-order, automation, and a reasonable expressive power. As an example, even if in principle type systems coming from light logics [13] indeed handle higher-order functions and can be easily implementable, the class of catched programs is small and full recursion is simply absent. On the other hand, Jost et al. [35] have successfully encapsulated Tarjan's amortised cost analysis into a type systems that allows a fully automatic resource analysis. In contrast to our work, only linear resource usage can be established. However, their cost metric is general, while our technique only works for time bounds. Also in the context of amortised analysis, Danielsson [21] provides a semiformal verification of the runtime complexity of lazy functional languages, which allows the derivation of non-linear complexity bounds on selected examples.

Term Rewriting. Traditionally, a major concern in rewriting has been the design of sound algorithmic methodologies for checking termination. This has given rise to many different techniques including basic techniques like path orders or interpretations, as well as sophisticated transformation techniques, c.f. [52, Chapter 6]. Complexity analysis of TRSs can be seen as a natural generalisation of termination analysis. And, indeed, variations on path orders and the interpretation methods capable of guaranteeing quantitative properties have appeared one after the other starting from the beginning of the nineties [7, 16, 38, 40]. In both termination and complexity analysis, the rewriting community has always put a strong emphasis to automation. However, with respect to higher-order rewrite systems (HRSs) only termination has received steady attention, c.f. [52, Chapter 11]. Except for very few attempts without any formal results complexity analysis of HRSs has been lacking [12, 17].

Cost Functions. An alternative strategy for complexity analysis consists in translating programs into other expressions (which could be programs themselves) whose purpose is precisely computing the complexity (also called the *cost*) of the original program. Complexity analysis is this way reduced to purely extensional reasoning on the obtained expressions. Many works have investigated this approach in the context of higher-order functional languages, starting from the pioneering work by Sands [49] down to more recent contributions, e.g. Vasconcelos et al. [53]. What is common among most of the cited works is that either automation is not considered (e.g. cost functions can indeed be produced, but the problem of putting them in closed form is not [53]), or the time complexity is not analysed parametrically on the size of the input [27]. A notable exception is Benzinger's work [14], which however only applies to programs extracted from proofs, and thus only works with primitive recursive definitions.

References

[1] B. Accattoli and U. Dal Lago. Beta Reduction is Invariant, Indeed. In *Proc. of 23rd CSL*, pages 8:1–8:10. ACM, 2014.

[2] B. Accattoli and C. Sacerdoti Coen. On the Usefulness of Constructors. In *Proc. of 30th LICS*. IEEE, 2015. To appear.

[3] E. Albert, S. Genaim, and A. N. Masud. On the Inference of Resource Usage Upper and Lower Bounds. *TOCL*, 14(3):22(1–35), 2013.

[4] T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *TCS*, 236(1–2):133–178, 2000.

[5] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A Program Logic for Resources. *TCS*, 389(3):411–445, 2007.

[6] M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 33–48, 2010.

[7] M. Avanzini and G. Moser. Polynomial Path Orders. *LMCS*, 9(4), 2013.

[8] M. Avanzini and G. Moser. Tyrolean Complexity Tool: Features and Usage. In *Proc. of 24th RTA*, volume 21 of *LIPIcs*, pages 71–80, 2013.

[9] M. Avanzini and G. Moser. A Combination Framework for Complexity. *IC*, 2015. To appear.

[10] M. Avanzini, U. Dal Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order (Long Version). *CoRR*, cs/CC/1506.05043, 2015. Available at `http://www.arxiv.org/abs/1506.05043`.

[11] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. ISBN 978-0-521-77920-3.

[12] P. Baillot and U. Dal Lago. Higher-Order Interpretations and Program Complexity. In *Proc. of 26th CSL*, volume 16 of *LIPIcs*, pages 62–76, 2012.

[13] P. Baillot and K. Terui. Light types for Polynomial Time Computation in Lambda Calculus. *IC*, 207(1):41–62, 2009.

[14] R. Benzinger. Automated Higher-order Complexity Analysis. *TCS*, 318(1-2):79–103, 2004.

[15] R. Bird. *Introduction to Functional Programming using Haskell, Second Edition*. Prentice Hall, 1998. ISBN 978-0-134-84346-9.

[16] G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with Polynomial Interpretation Termination Proof. *JFP*, 11(1):33–53, 2001.

[17] G. Bonfante, J.-Y. Marion, and R. Péchoux. Quasi-interpretation Synthesis by Decomposition and an Application to Higher-order Programs. In *Proc. of 4th ICTAC*, volume 4711 of *LNCS*, pages 410–424, 2007.

[18] U. Dal Lago and M. Gaboardi. Linear Dependent Types and Relative Completeness. *LMCS*, 8(4), 2012.

[19] U. Dal Lago and S. Martini. On Constructor Rewrite Systems and the Lambda Calculus. *LMCS*, 8(3):1–27, 2012.

[20] U. Dal Lago and B. Petit. The Geometry of Types. In *Proc. of 40th POPL*, pages 167–178. ACM, 2013.

[21] N. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proc. of 35th POPL*, pages 133–144. ACM, 2008.

[22] O. Danvy and L. R. Nielsen. Defunctionalization at Work. In *Proc. of 3rd PPDP*, pages 162–174. ACM, 2001.

[23] G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *JAR*, 33(3-4):341–383, 2004.

[24] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and Disproving Termination of Higher-Order Functions. In *Proc. of 5th FROCOS*, volume 3717 of *LNCS*, pages 216–231, 2005.

[25] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *TOPLAS*, 33(2):7:1–7:39, 2011.

[26] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving Termination of Programs Automatically with AProVE. In *Proc. of 7th IJCAR*, volume 8562 of *LNCS*, pages 184–191, 2014.

[27] G. Gomez and Y. Liu. Automatic Time-bound Analysis for a Higher-order Language. In *Proc. of 9th PEPM*, pages 75–86. ACM, 2002.

[28] B. Gramlich. Abstract Relations between Restricted Termination and Confluence Properties of Rewrite Systems. *FI*, 24:3–23, 1995.

[29] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. ISBN 978-1-107-02957-6.

[30] N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR*, volume 5195 of *LNAI*, pages 364–380, 2008.

[31] N. Hirokawa and G. Moser. Complexity, Graphs, and the Dependency Pair Method. In *Proc. of 15th LPAR*, volume 5330 of *LNCS*, pages 652–666, 2008.

[32] N. Hirokawa, A. Middeldorp, and H. Zankl. Uncurrying for Termination and Complexity. *JAR*, 50(3):279–315, 2013.

[33] N. D. Jones. Flow Analysis of Lazy Higher-order Functional Programs. *TCS*, 375(1-3):120–136, 2007.

[34] N. D. Jones and N. Bohr. Call-by-Value Termination in the Untyped lambda-Calculus. *LMCS*, 4(1), 2008.

[35] S. Jost, K. Hammond, H.-W. Loidl, and M.Hofmann. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proc. of 37th POPL*, pages 223–236. ACM, 2010.

[36] J. Kochems and L. Ong. Improved Functional Flow and Reachability Analyses Using Indexed Linear Tree Grammars. In *Proc. of 22nd RTA*, volume 10 of *LIPIcs*, pages 187–202, 2011.

[37] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 295–304, 2009.

[38] J.-Y. Marion. Analysing the Implicit Complexity of Programs. *IC*, 183: 2–18, 2003.

[39] J. Midtgaard. Control-flow Analysis of Functional Programs. *ACM Comput. Surv.*, 44(3):10, 2012.

[40] G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, cs.LO/0907.5527, 2009. Habilitation Thesis.

[41] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005. ISBN 978-3-540-65410-0.

[42] L. Noschinski, F. Emmes, and J. Giesl. A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems. In *Proc. of 23rd CADE*, LNAI, pages 422–438. Springer, 2011.

[43] C. Okasaki. Functional Pearl: Even Higher-Order Functions for Parsing. *JFP*, 8(2):195–199, 1998.

[44] S. E. Panitz and M. Schmidt-Schauß. TEA: Automatically Proving Termination of Programs in a Non-Strict Higher-Order Functional Language. In *Proc. of 4th SAS*, pages 345–360, 1997.

[45] B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

[46] G. D. Plotkin. LCF Considered as a Programming Language. *TCS*, 5 (3):223–255, 1977.

[47] F. Rabhi and G. Lapalme. *Algorithms: A Functional Programming Approach*. Addison-Wesley, 1999. ISBN 978-0-201-59604-5.

[48] J. C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[49] D. Sands. Complexity Analysis for a Lazy Higher-Order Language. In *Proc. of 3rd ESOP*, volume 432 of *LNCS*, pages 361–376, 1990.

[50] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Proc. of 26th CAV*, volume 8559 of *LNCS*, pages 745–761, 2014.

[51] C. Sternagel and R. Thiemann. Generalized and Formalized Uncurrying. In *Proc. of 8th FROCOS*, volume 6989 of *LNCS*, pages 243–258, 2011.

[52] TeReSe. *Term Rewriting Systems*, volume 55. Cambridge University Press, 2003. ISBN 978-0-521-39115-3.

[53] P. B. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Revised Papers of 15th Workshop on IFL*, pages 86–101, 2003.

[54] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The Worst Case Execution Time Problem - Overview of Methods and Survey of Tools. *TECS*, pages 1–53, 2008.

[55] H. Zankl and M. Korp. Modular Complexity Analysis for Term Rewriting. *LMCS*, 10(1:19):1–33, 2014.