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

Martin Avanzini  Ugo Dal Lago  Georg Moser
Università di Bologna, Italy & INRIA, France  University of Innsbruck, Austria
martin.avanzini@uibk.ac.at  ugo.dallago@unibo.it  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 non-trivial 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 required to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Permission © 2015 ACM 978-1-4503-3669-7/15/08 . . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

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 have been proved for various forms of rewrite systems, including the λ-calculus and interactive theorem provers; see [1, 5, 35] 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 the unitary cost model, a program is attributed a cost equal to the number 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.

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 have been proved for various forms of rewrite systems, including the λ-calculus and interactive theorem provers; see [1, 5, 35]. 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 (sometimes even by a linear function) 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 systems, a formal model of computation that is at the heart of functional programming. Examples are AProVE, CaR, and ProVERB. These first-order provers (FOPs for short) combine many different techniques, and after some years of development,
start being able to treat non-trivial programs, as demonstrated by the result of the annual termination competition. This is potentially very interesting also for the complexity analysis of higher-order functional programs, since well-known transformation techniques such as defunctionalisation are available, which turn higher-order functional programs into equivalent first-order ones. This has been done in the realm of termination, but appears to be infeasible in the context of complexity analysis. Conclusively this program transformation approach has been reflected critical in the literature, cf. [55].

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 programs having a recursive structure too complicated for FOPs to handle 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. In all cases we can automatically verify the effectiveness of our approach, as at least termination of the original program can be detected fully automatically. For all but five cases, we achieve a fully automated complexity analysis, which is almost always asymptotically optimal. As far as we know, no other automated complexity tool can handle the five open examples.

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.

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:

```ocaml
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:

```ocaml
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 *)
```

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 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. 

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

http://termination-portal.org/wiki/Termination_Competition
More precisely, an expression of type \( \text{'a cl} \) represents a function closure, whose arguments are used to store assignments to free variables. An infix operator \((\varnothing)\), modelling application, can then be defined as follows:\(^2\)

\[
\text{let rec } (\varnothing) \text{ cl z } = \\
\text{let rec cl with} \\
\text{Cl}_1(f,g) \rightarrow f \varnothing (g \varnothing z) \\
\mid \text{Cl}_2 \rightarrow z \\
\mid \text{Cl}_3(x) \rightarrow x::z;;
\]

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

\[
\text{let comp } f \ g = \text{Cl}_1(f,g) ;; \\
\text{let rec walk xs } = \\
\text{match xs with} \\
[ ] \rightarrow \text{Cl}_2 \\
| x::ys \rightarrow \text{comp } (\text{walk } ys) \text{ Cl}_3(x) ;; \\
\text{let rev l } = \text{walk } l \ \varnothing [ ]; ;; \\
\text{let main l } = \text{rev } l ;;
\]

Observe that now the recursive function \( \text{walk} \) constructs an explicit representation of the closure computed by its original definition. The function \((\varnothing)\) 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 7] for a formal treatment. On our running example, this program transformation results in the rewrite system \( A_{\text{rev}} \), which looks as follows:\(^3\)

\[
\begin{align*}
\text{Cl}_1(f,g) \varnothing z & \rightarrow f \varnothing (g \varnothing z) \\
\text{Cl}_2 \varnothing z & \rightarrow z \\
\text{Cl}_3(x) \varnothing z & \rightarrow x::z \\
\text{comp } f \varnothing g & \rightarrow \text{Cl}_1(f,g) \\
\text{comp } f \varnothing f & \rightarrow \text{comp } (f) \\
\text{match } \text{walk } [ ] & \rightarrow \text{Cl}_2 \\
\text{match } \text{walk } (x::ys) & \rightarrow \\
\text{comp } \varnothing (\text{fix } \text{walk } \varnothing ys) \varnothing \text{Cl}_3(x) \\
\text{walk } \varnothing xs & \rightarrow \text{match } \text{walk } (xs) \\
\text{fix } \text{walk } \varnothing xs & \rightarrow \text{walk } xs \\
\text{rev } \varnothing l & \rightarrow \text{fix } \text{walk } \varnothing l \varnothing [ ] \\
\text{main } l & \rightarrow \text{rev } l \\
\end{align*}
\]

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 \(^{30},^{32}\)) 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 \((\varnothing)\), thus highlighting the deep recursive structure of the program above? Let us, for example, focus on the rewriting rule

\[
\text{Cl}_1(f,g) \varnothing z \rightarrow f \varnothing (g \varnothing z),
\]

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 where \(g\) is of the form \(\text{Cl}_3(x)\), and \(f\) is either \(\text{Cl}_2\) or again of the form \(\text{Cl}_1(f,g)\). 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. \(\text{comp}\) or \(\text{match}_\text{walk}\) 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 \( R_{\text{rev}} \), which is precisely what \( R_\text{DC} \) produces in output:

\[
\begin{align*}
\text{Cl}_1(\text{Cl}_2,\text{Cl}_3(x),z) & \rightarrow x::z \\
\text{Cl}_1(\text{Cl}_1(f,g),\text{Cl}_3(x),z) & \rightarrow \text{Cl}_1(f,g,x::z) \\
\text{fix}^{\text{walk}}()[ ] & \rightarrow \text{Cl}_2 \\
\text{fix}^{\text{walk}}(x::ys) & \rightarrow \text{Cl}_1(\text{fix}_1^{\text{walk}}(ys),\text{Cl}_3(x)) \\
\text{main }()[ ] & \rightarrow [ ] \\
\text{main } (x::ys) & \rightarrow \text{Cl}_1(\text{fix}_1^{\text{walk}}(ys),\text{Cl}_3(x),[ ])
\end{align*}
\]

This term rewrite system is equivalent to \( A_{\text{rev}} \) from above, both extensionally and in terms of the underlying complexity. However, the FOPs we have considered can indeed conclude that \( \text{main} \) 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 \( A_{\text{rev}} \) into \( R_{\text{rev}} \). Before, we recap central definitions in the next section.

3. Preliminaries

The purpose of this section is to give some preliminary notions about the \( \lambda \)-calculus, term rewrite systems, and translations between them; see \(^{10},^{15},^{52}\) for further reading.

To model a reasonable rich but pure and monomorphic functional language, we consider a typed \( \lambda \)-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 \( C_1, \ldots, C_k \) be finitely many constructors, each equipped with a fixed arity. The syntax of PCF-programs is given by the following grammar:

\[
\text{Exp } e,f ::= x \mid C_i(e) \mid \lambda x.e \mid e \mid \text{fix}(x.e) \\
| \text{match } e \{ C_1(z_1) \mapsto e_1; \ldots; C_k(z_k) \mapsto e_k \},
\]

---

\(^2\) The definition is rejected by the OCaml type-checker, which however, is not an issue in our context.

\(^3\) In \( A_{\text{rev}} \) rule \(^9\) reflects that, under the hood, we treat recursive let expressions as syntactic sugar for a dedicated fixpoint operator.
where $x$ ranges over variables. Note that the variables $\bar{x}$ in a match-
expression are considered bound in $e$. A simple type system can be
easily defined 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.

We adopt weak call-by-value semantics. Here weak means
that reduction under any $\lambda$-abstraction $\lambda x.e$ and any fixpoint-
expressions $\text{fix}(x.e)$ is prohibited. The definition is straightforward, see e.g. [29]. Call-by-value means that in a redex $e$, the expression $f$ has to be evaluated. A match-expression match $e\{G_1(\bar{x}) \rightarrow e_1; \ldots; G_n(\bar{x}) \rightarrow e_n\}$ is evaluated by first evaluating the guard $e$ to a value $C_i(\bar{v})$. Reduction then continues with the corresponding case-expression $e_i$ with values $v_i$ substituted for variables $\bar{x}_i$. The one-step weak call-by-value reduction relation is denoted by $\rightarrow_w$. Elements of the term algebra over constructors $C_1, \ldots, C_n$ embed-
ded in our language are collected in Input. A PCF-program with
$n$ input arguments is a closed expression $P = \lambda z_1 \cdots \lambda z_n.e$ 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. 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]). In this way we also ensure that the abstractions re-
duced in a run of $P$ are the ones found in $P$, an essential property
for performing functionalisation. We assume that variables in $P$
have been renamed apart, and we impose a total order on variables
in $P$. The free variables $\text{FV}(e)$ in the body $e$ of $P$ can this way be
defined as an ordered sequence of variables.

Example 1. We fix constructors $\square$ and $(\cdot : \cdots)$ 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_{rev} := \lambda l. \text{rev}
\begin{align*}
\text{rev} & := \lambda l. \text{fix}(w. \text{walk}
\begin{align*}
\text{walk} & := \lambda x s. \text{match}
\begin{align*}
\text{comp} & := \lambda f. \lambda \bar{x}. \lambda \bar{z}. f(\bar{g}. \bar{z})
\end{align*}
\end{align*}
\end{align*}

The second kind of functional formalism we will deal with is
the one of term rewrite systems. Let $\mathcal{F} = \{e_1, \ldots, e_k\}$ be a set of function symbols, each equipped again with an arity,
the signature. We denote by $\sigma_t$ the 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: $t|_p = t$ for the empty position $\varepsilon$, and $t|_{tp} = t|_p$ for
$t = t_1, \ldots, t_q$. For a position $p$ in $t$, we denote by $t|_p$ 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
$\square$, the hole. We define $C[\varepsilon] := C[\varepsilon]$ for $p$ the position of $\square$ in $C$,
i.e. $C[p] = \emptyset$.

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

A term rewrite system (TRS for short) $\mathcal{R}$ is a finite set of rewrite
rules, i.e. directed equations $f(t_1, \ldots, t_q) \rightarrow r$ such that all variables
occurring in the right-hand side $r$ occur also in the left-hand
side $f(t_1, \ldots, t_q)$. 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 $T(\mathcal{C}_\mathcal{R})$.
We denote by $\rightarrow_\mathcal{R}$ the one-step rewrite relation of $\mathcal{R}$, imposing
call-by-value semantics. Call-by-value means that variables are as-
signed elements of $T(\mathcal{C}_\mathcal{R})$. Throughout the following, we consider
non-ambiguous rewrite systems, that is, the left-hand sides are pair-
wise non-overlapping. Despite the restriction to non-ambiguous
rewrite systems the rewrite relation $\rightarrow_\mathcal{R}$ may be non-deterministic:
eg. 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 $\mathcal{R}$, all normalising reductions of $t$ have the same length, i.e. if $t \rightarrow^*_\mathcal{R} u_1$ and $t \rightarrow^*_\mathcal{R} u_2$
for two irreducible terms $u_1$ and $u_2$, then $u_1 = u_2$ and $n = n$.

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 ($\times$),
the application symbol. Here, we are more liberal and just assume the
presence of $\times$ (and), and allow function symbols that take $m$ and
the one argument. Throughout the following, we are foremost dealing with
ATRSs, which we denote by $A, B$ below. We also write $(\times)$"
and assume that it associates to the left.

In the following, we show that every PCF-program $P$ can be seen as
an ATRS $A_P$. To this end, we first define an infinite schema $A_{PCF}$
of rewrite rules which allows us to evaluate the whole of PCF. The
signature underlying $A_{PCF}$ contains, besides the application-symbol
($\times$) and constructors $C_1, \ldots, C_n$, the following function symbols,
called closure constructors: (i) for each PCF term $\lambda x.e$ with $n$ free
variables an $n$-ary symbol $\lambda_{a,x,e}$; (ii) for each PCF term $\text{fix}(x.e)$
with $n$ free variables an $n$-ary symbol $\text{fix}_{a,x,e}$; and (iii) for each
match-expression match $e\{c\}$ with $n$ free variables a symbol
$\text{match}_{a,c}$. Hence, for each $\text{PCF}$ term $\text{from}$ $A_{PCF}$ terms as follows.

\[
\begin{align*}
[x] & := x; \\
\lambda x.e & := \lambda_{a,x,e}(\bar{x}), \text{where } \bar{x} = \text{FV}(\lambda x.e); \\
C(e_1, \ldots, e_k) & := C_i(e_1, \ldots, e_k); \\
\text{fix}_{a,c} & := \text{fix}_{a,x,e}(\bar{c}), \text{where } \bar{c} = \text{FV}(\text{fix}(x.e)); \\
\text{match}_{a,c} & := \text{match}_{a,c} \{ e_{i} \bar{x}, \bar{c} \}, \text{where } \bar{c} = \text{FV}(\text{fix}(x.e))
\end{align*}
\]

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

\[
\begin{align*}
\lambda_{a,x,e} & := \lambda_{a,x,e}(\bar{x}) \times x \rightarrow [e]\times; \\
\text{fix}_{a,c}(\bar{x}) & := \text{fix}_{a,x,e}(\bar{c}) \times y \rightarrow [e \text{fix}(x.e)/x]\times y, \text{where } y \text{ is fresh}; \\
\text{match}_{a,c}(\bar{x}, \bar{c}) & := [e_{i}\bar{x}, \bar{c}], \text{for } i \in \{1, \ldots, k\}
\end{align*}
\]

Here, we suppose $c_{\text{es}} = \{ C_i(\bar{x}) \rightarrow e_1; \ldots; C_n(\bar{x}) \rightarrow e_n \}$.

For a program $P = \lambda_1 \cdots \lambda_n.e$, we define $A_P$ as the least set of
rules that (i) contains a rule $\text{main}(x_1, \ldots, x_n) \rightarrow [e]\times$, where
$\text{main}$ is a dedicated function symbol; and (ii) whenever $t \rightarrow r \in A_P$
and $f$ is a closure-constructor in $r$, then $A_P$ contains all defining
rules of $f$ from the schema $A_{PCF}$. Crucial, $A_P$ is always finite, in
fact, the size of $A_P$ is linearly bounded in the size of $P$ [10].

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

\[
\begin{align*}
\text{fix}_{a,c}(\bar{x}) @ y & \rightarrow [e][x](\text{fix}_{a,c}(\bar{x}) @ y),
\end{align*}
\]

which would also correctly model the semantics of fixpoints $\text{fix}(x.e)$.

Then the closure constructors occurring in $A_P$ 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 $A_P$. 

Example 2. The expression $P_{\text{rev}}$ from Example 1 gets translated into the ATRS $\mathcal{A}_{\text{rev}} = \mathcal{A}_{\text{rev}}$ 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 $\rightarrow_n$-reduction of an expression $P d_1 \cdots d_n$ ($d_i \in \text{Input}$) is simulated step-wise by a call-by-value $A_P$-derivation starting from $\text{main}(d_1, \ldots, d_n)$.

As the inverse direction of this proposition can also be stated, $A_P$ 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 $d \in \text{Input}$ 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 $\text{main}(d_1, \ldots, d_n)$. 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 $\sum_{i=1}^n |d_i| \leq m$, 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 $A_P$ 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 as a Pyrrhic victory towards our final goal: as discussed in Section 3, 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, 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 $f(\mathcal{A})$ 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 $A$. 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 relaxed 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 $\mathcal{A}_{\text{rev}}$ from Section 2. There, for instance, the call to $\text{walk}$ in the definition of $\text{fix}_{\text{walk}}$ could be inlined, thus resulting in a new definition:

$$\text{fix}_{\text{walk}} @ xs \rightarrow \text{match}_{\text{walk}}(xs).$$

Informally, thus, inlining consists in modifying the right-hand-sides of ATRS rules by rewriting subtrees, 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, $\text{match}_{\text{walk}}$ cannot be inlined immediately, simply because $\text{match}_{\text{walk}}$ is defined itself by case analysis on $xs$. To allow inlining of this function nevertheless, we specialize $xs$ to the patterns $[]$ and $x::ys$, the patterns underlying the case analysis of $\text{match}_{\text{walk}}$. This results in two alternative rules for $\text{fix}_{\text{walk}}$, namely

$$\text{fix}_{\text{walk}} @ [] \rightarrow \text{match}_{\text{walk}}([],) : \text{fix}_{\text{walk}} @ (x::ys) \rightarrow \text{match}_{\text{walk}}(x::ys).$$

Now we can inline $\text{match}_{\text{walk}}$ in both rules. As a consequence the rules defining $\text{fix}_{\text{walk}}$ 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 $s \overset{\mu}{\rightarrow}_A t$, if there exists a rule $l \rightarrow r \in \mathcal{A}$ such that $\mu$ is a unifier of left-hand side $l$ and the subterm $s_p$ (after renaming apart variables in $l \rightarrow r$ and $s$) and $t = s[\mu](p)$. In other words, the instance $s_p$ of $s$ rewrites to $t$ at position $p$ with rule $l \rightarrow r \in \mathcal{A}$. The substitution $\mu$ is just enough to uncover the corresponding redex in $s$. Note, however, that the performed rewrite step is not necessarily call-by-value, the mu $\mu$ could indeed contain function calls. We define the set of all inlinings of a rule $l \rightarrow r$ at position $p$ which is labeled by a defined symbol by

$$\text{inline}_{\mathcal{A},\mu}(l \rightarrow r) := \{ l \mu \rightarrow s | r \overset{\mu}{\rightarrow}_A s \} .$$

The following example demonstrates inlining through narrowing.

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

$$\text{match}_{\text{walk}}(xs) \overset{\mu_1}{\rightarrow}_{\text{rev}} \text{Cl}_2$$

$$\text{match}_{\text{walk}}(xs) \overset{\mu_2}{\rightarrow}_{\text{rev}} \text{comp } \circ (\text{fix}_{\text{walk}}(xs)) \circ \text{Cl}_3(xs).$$

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

$$\text{inline}_{\mathcal{A}_{\text{rev}},\mu_1}(\text{fix}_{\text{walk}} @ xs \rightarrow \text{match}_{\text{walk}}(xs))$$

consists of the two rules

$$\text{fix}_{\text{walk}} @ [] \rightarrow \text{Cl}_2;$$

$$\text{fix}_{\text{walk}} @ (x::ys) \rightarrow \text{comp } \circ (\text{fix}_{\text{walk}} @ ys) \circ \text{Cl}_3(xs).$$

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. $k(x,y) \rightarrow x$
2. $\text{main}(0) \rightarrow 0$
3. $\text{main}(S(n)) \rightarrow k(\text{main}(n),\text{main}(n))$

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

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

\textbf{Example 5.} Consider the system consisting of the following rules.

\begin{align*}
1. \quad & h(x, 0) \rightarrow x \\
2. \quad & \text{main}(0) \rightarrow 0 \\
3. \quad & \text{main}(S(n)) \rightarrow h(\text{main}(n), n)
\end{align*}

Inlining $h$ in the definition of main will specialise the variable $n$ to 0 and thus replaces rule 3 by $\text{main}(S(n)) \rightarrow h(\text{main}(n), 0)$. 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 \textit{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 \textit{sufficiently defined}, with respect to an ATRS $\mathcal{A}$, if all subterms $f(t)$ occurring in a reduction of $\text{main}(d_1, \ldots, d_n)$ ($d_j \in \text{Input}$) are reducible. This property is not decidable in general. Still, the ATRSs obtained from the translation in Section 6 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 $\leadsto_{\mathcal{A}}$ an extension of $\rightarrow_{\mathcal{A}}$ where not all arguments are necessarily reduced, but where still a step cannot delete redexes: $s \leadsto_{\mathcal{A}} t$ if $s = C[l\sigma]$ and $t = C[r\sigma]$ for a context $C$, rule $l \rightarrow r \in \mathcal{A}$ and a substitution $\sigma$ which satisfies $\sigma(x) \in \mathcal{C}(\mathcal{A})$ for all variables $x$ which occur in $l$ but not in $r$. By definition, $\rightarrow_{\mathcal{A}} \subseteq \leadsto_{\mathcal{A}}$. The relation $\leadsto_{\mathcal{A}}$ 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 $l \rightarrow r$ with an application of a corresponding rule in $\text{inline}_{\mathcal{A},p}(l \rightarrow r)$. 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.

\textbf{Lemma 1.} Let $l \rightarrow r$ 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 $\text{main}(d_1, \ldots, d_n) \rightarrow_{\mathcal{A}} C[l\sigma] \rightarrow_{\mathcal{A}} C[r\sigma] \rightarrow_{\mathcal{A}} u$, for $d_j \in \text{Input}$ ($i = 1, \ldots, n$) and some $\ell \in \mathbb{N}$. Then there exists a term $t$ such that the following properties hold:

1. $l\sigma \rightarrow_{\text{inline}_{\mathcal{A},p}(l \rightarrow r)} t$; and
2. $r\sigma \leadsto_{\mathcal{A}} t$, where $A$ collects all rules that are unifiable with the right-hand side $r$ at position $p$; and
3. $C[t] \rightarrow_{\mathcal{A}}^{\omega} C[t]^\omega u$.

In consequence, we thus obtain a term $t$ such that $\text{main}(d_1, \ldots, d_n) \rightarrow_{\mathcal{A}} C[l\sigma] \rightarrow_{\text{inline}_{\mathcal{A},p}(l \rightarrow r)} C[t] \rightarrow_{\mathcal{A}}^{\omega} C[t]^\omega u$, 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 $\leadsto_{\mathcal{A}}$ instead of $\rightarrow_{\mathcal{A}}$. To ensure that the induction is well-defined, we use the following specialization of [28] Theorem 3.13).

\textbf{Proposition 3. If a term $t$ has a normalform wrt. $\rightarrow_{\mathcal{A}}$, then all $\leadsto_{\mathcal{A}}$ reductions of $t$ are finite.}

\textbf{Theorem 1.} Let $l \rightarrow r$ 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 $l \rightarrow r$ by the rules $\text{inline}_{\mathcal{A},p}(l \rightarrow r)$. Then every normalizing derivation with respect to $\mathcal{A}$ starting from $\text{main}(d_1, \ldots, d_n)$ ($d_j \in \text{Input}$ of length $\ell$ is simulated by a derivation with respect to $\mathcal{B}$ from $\text{main}(d_1, \ldots, d_n)$ of length at least $\lceil \frac{\ell}{2} \rceil$.

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

We define the derivation height $\text{dh}(s)$ of a term $s$ wrt. the relation $\leadsto_{\mathcal{A}}$ as the maximal $m$ such that $s \leadsto_{\mathcal{A}}^m u$ holds. The proof is by induction on $\text{dh}(t)$, which is well-defined by assumption and Proposition 3. It suffices to consider the induction step. Suppose $t \rightarrow_{\mathcal{A}} s \rightarrow_{\mathcal{A}} u$. We consider the case where the step $t \rightarrow_{\mathcal{A}} s$ is obtained by applying the rule $l \rightarrow r \in \mathcal{A}$, otherwise, the claim follows directly from induction hypothesis. Then as a result of Lemma 1 and 13, we obtain an alternative derivation

$t \rightarrow_{\mathcal{A}} u \rightarrow_{\mathcal{B}} u$,

for some term $t$ and $u \rightarrow_{\mathcal{B}} u$. Note that $s \leadsto_{\mathcal{A}} u$ as a consequence of Lemma 13, and thus $\text{dh}(s) > \text{dh}(u)$ by definition of derivation height. Induction hypothesis on $u$ thus yields a derivation $t \rightarrow_{\mathcal{A}} u \rightarrow_{\mathcal{B}} u$ of length at least $\lceil \frac{\ell}{2} \rceil + 1 = \lceil \frac{\ell + 2}{2} \rceil \geq \lceil \frac{\ell}{2} \rceil$.

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

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

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

\begin{align*}
1. \quad & C_{\text{inl}}(f, g) \rightarrow f \circ (g \circ z) \\
2. \quad & C_{\text{inl}} @ z \rightarrow z \\
3. \quad & C_{\text{inl}}(x) \rightarrow x : z \\
4. \quad & \text{comp}_1(f) \rightarrow \text{comp}_1(f) \\
5. \quad & \text{comp } @ f \rightarrow \text{comp}_1(f) \\
6. \quad & \text{match}_{\text{walk}}([1]) \rightarrow C_{\text{inl}} \\
7. \quad & \text{match}_{\text{walk}}(x : y) \rightarrow \text{comp } @ (\text{fix}_{\text{walk}} @ y) @ C_{\text{inl}}(x)
\end{align*}
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 $A_{\text{rev}}$.

Note that the modified system from Example 6 gives further possibilities for inlining. For instance, we could narrow further down the call to $\text{fix}_\text{walk}$ 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 $\text{fix}_\text{walk}$, ad infinitum. We could have also further reduced the rules defining $\text{match}_\text{walk}$ and $\text{walk}$. However, it is not difficult to see that these rules will never be unfolded in a call to $\text{main}$, 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 and complexity analysis), 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 $l \rightarrow r \in A$ *usable* if it can be applied in a derivation

$$
\text{main}(d_1, \ldots, d_k) \rightarrow_A l \rightarrow_A 1 \rightarrow_A \ldots \rightarrow_A t_1 \rightarrow_1 \rightarrow_A t_2 .
$$

where $d_i \in \text{Input}$. The rule $l \rightarrow r$ 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 $A$ to a subset of $A$ 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 $\text{main}$, and remove the defining rules of symbols not in this collection, compare e.g. [33]. 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 4.3 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 2. 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 $\text{main}(l)$. In particular, none of the symbols $\text{rev}$, $\text{walk}$, $\text{comp}$ and $\text{comp}_1$ are passed to $\text{Cl}_1$. With this knowledge, it is not difficult to see that their defining rules, together with the rules defining $\text{match}_\text{walk}$, 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.

### 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 6, 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. 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 $\text{Cl}_1$.

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 $s \rightarrow t$ an *instance* of a rule $l \rightarrow r$, if there is a substitution $\sigma$ with $s = l\sigma$ and $t = r\sigma$.

We say that an ATRS $B$ is an *instantiation of* $A$ iff all rules in $B$ are instances of rules from $A$.

This instantiation is *sufficiently exhaustive* if for every derivation

$$
\text{main}(d_1, \ldots, d_k) \rightarrow_A t_1 \rightarrow_A t_2 \rightarrow_A \ldots,
$$

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

$$
\text{main}(d_1, \ldots, d_k) \rightarrow_B t_1 \rightarrow_B t_2 \rightarrow_B \ldots
$$

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 $\text{Cl}_1(f,g) @ x \rightarrow f @ (g @ x)$ by the two rules

$$
\text{Cl}_1(\text{Cl}_2, \text{Cl}_3(x)) @ x \rightarrow \text{Cl}_3(x) @ (\text{Cl}_2 @ z)
$$

$$
\text{Cl}_1(\text{Cl}_1(f,g), \text{Cl}_3(x)) @ z \rightarrow \text{Cl}_1(f,g) @ (\text{Cl}_1 @ z)
$$

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 $\text{main}(l)$ for a list $l$, arguments to $\text{Cl}_1$ 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:

$$
\text{Cl}_2 @ z \rightarrow z
$$

---

1. $\text{Cl}_1 @ z \rightarrow z$
2. $\text{Cl}_2 @ z \rightarrow z$
3. $\text{Cl}_3(x) @ z \rightarrow x::z$
4. $\text{fix}_\text{walk} @ [] \rightarrow \text{Cl}_2$
5. $\text{fix}_\text{walk} @ (x::ys) \rightarrow \text{Cl}_1(\text{fix}_\text{walk} @ ys, \text{Cl}_3(x))$
6. $\text{main}(l) \rightarrow \text{fix}_\text{walk} @ l @ []$
2 \text{Cl}_1(\text{Cl}_2, \text{Cl}_3(x)) \otimes z \rightarrow x::z

3 \text{Cl}_1(\text{Cl}_1(f, g), \text{Cl}_3(x)) \otimes z \rightarrow
\text{Cl}_1(\text{fix}_\text{walk}, \text{Cl}_3(x)) \otimes z

4 \text{fix}_\text{walk}([\text{fix}_\text{walk}, \text{Cl}_3(x)]) \rightarrow \text{Cl}_2

5 \text{fix}_\text{walk}((x::ys), \text{Cl}_3(x)) \rightarrow
\text{Cl}_1(\text{fix}_\text{walk} \otimes ys, \text{Cl}_3(x)) \otimes z.

One can then check that the resulting system is \(\eta\)-saturated.

**Lemma 2.** Let \(A_\eta\) be the \(\eta\)-saturation of \(A\).

1. The rewrite relation \(\rightarrow_{A_\eta}\) coincides with \(\rightarrow_{A}\).
2. Suppose \(A_\eta\) is head variable free. If \(s \rightarrow_{A_\eta} t\) then \(s.s.a \rightarrow_{A_\eta}\).

**Proof.** For Property 1 the inclusion \(\rightarrow_{A} \subseteq \rightarrow_{A_\eta}\) follows trivially from the inclusion \(\rightarrow_{A} \subseteq \rightarrow_{A_\eta}\). The inverse inclusion \(\rightarrow_{A_\eta} \subseteq \rightarrow_{A}\) can be shown by a standard induction on the derivation of \(l \rightarrow r\) in \(A_\eta\).

Property 2 can be proven by induction on \(t\). The proof follows the pattern of the proof of Sterngel and Thiemann [51]. Notice that in [51] Theorem 10, the rewrite system \(\omega_{A_\eta}\) is enriched with uncourrying rules of the form \(f^i(x_1, \ldots, x_n) \circ y \rightarrow f^{i+1}(x_1, \ldots, x_n, y)\). Such an extension is not necessary in the absence of head variables. In our setting, the application symbol is completely eliminated by uncourrying, and thus the above rules are dead code. \(\square\)

As a consequence, we immediately obtain the following theorem.

**Theorem 3** (Uncourrying Transformation). Suppose that \(A\) is head variable free. The uncourrying transformation, which maps an ATRS \(A\) to the system \(\omega_{A_\eta}\), is complexity reflecting.

Example 10 (Continued from Example 9). Uncourrying the \(\eta\)-saturated ATRS, consisting of the six rules from Example 9 and the two rules from Example 9 results in the following set of rules:

1 \text{Cl}_1(\text{Cl}_2, \text{Cl}_3(x), z) \rightarrow x::z

2 \text{Cl}_1(\text{Cl}_1(f, g), \text{Cl}_3(x), z) \rightarrow \text{Cl}_1(f, g, x::z)

3 \text{Cl}_1(z) \rightarrow z

4 \text{fix}_\text{walk}([\text{fix}_\text{walk}, \text{Cl}_3(x)]) \rightarrow \text{Cl}_2

5 \text{fix}_\text{walk}((x::ys), \text{Cl}_3(x)) \rightarrow \text{Cl}_1(\text{fix}_\text{walk} \otimes ys, \text{Cl}_3(x))

6 \text{fix}_\text{walk}([\text{fix}_\text{walk}, \text{Cl}_3(x)]) \rightarrow \text{Cl}_1(z)

7 \text{fix}_\text{walk}((x::ys), z) \rightarrow \text{Cl}_1(\text{fix}_\text{walk} \otimes ys, \text{Cl}_3(x), z)

8 \text{main}(l) \rightarrow \text{fix}_\text{walk}^2(l, [\text{fix}_\text{walk}, \text{Cl}_3(x)])

Inlining the calls to \text{fix}\_walk and \text{Cl}\_3(x), followed by dead code elimination, results finally in the TRS \(\mathcal{R}_{\text{rec}}\) from Section 4.

### 5. Automation

In the last section we have laid the formal foundation of our program transformation methodology, and ultimately of our tool \text{HJAVA}. Up to now, however, program transformations (except for uncourrying) 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 → r is sensible at the current stage in our transformation pipeline. The algorithm inline(P) replaces every rule l → r by inline_{ATRS}(l → r) for some position p such that P(p, l → 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 labelled at position p by a symbol of the form match_. That is, the predicate enables inlining of calls resulting from the translation of a match-expression, thereby eliminating one induction due to the encoding of pattern matching during defunctionalisation.

• lambda-rewrite: This predicate holds if the subterm |l| of the form lambda_{f} \( \theta \) is of the form \( \text{match}_{i} \). 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 \( \text{const} \). This way, we avoid that variables f and \( \theta \) are improperly instantiated.

• constructor: The predicate holds if the right-hand sides of all rules used to inline \( r|_{i} \) 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 \( r|_{i} \) constitutes the only call-site to the inlined function f. This way, all rules defining f in A will turn to dead code after inlining.

(ii) size decreasing: each right-hand side in inline_{ATRS}(l → 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 \( A_{ow} \) 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 \( A = \{l_{i} \rightarrow r_{i}\}_{i \in \{1,\ldots,n\}} \). We define the collecting semantics of \( A \) as a tuple \( (Z_{1},\ldots,Z_{n}) \), where

\[ Z_{i} := \{(\sigma, t) | \exists \overline{d} \in \text{Input} \}. \]

\[ \text{main}(\overline{d}) \rightarrow_{A_{i}} C[\sigma] \rightarrow_{A} C[r_{s} \sigma] \text{ and } r_{s} \sigma \rightarrow_{A} t. \]

Here the substitutions \( \sigma \) are restricted to the set \( \text{Var}(l_{i}) \) of variables occurring in the left-hand side in \( l_{i} \).

The collecting semantics of \( A \) includes all the necessary information for implementing both dead code elimination and instantiation:

**Lemma 3.** The following properties hold:

1. The rule \( l_{i} \rightarrow r_{i} \in A \) constitutes dead code if and only if \( Z_{i} = \emptyset \).

2. Suppose the ATRS \( B \) is obtained by instantiating rules \( l_{i} \rightarrow r_{i} \) with substitutions \( \sigma_{i,1},\ldots,\sigma_{i,k} \). Then the instantiation is sufficiently exhaustive if for every substitution \( \sigma \) with \( (\sigma, t) \in Z_{i} \), there exists a substitution \( \sigma_{i,j} (j \in \{1,\ldots,k\}) \) which is at least as general as \( \sigma \).

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

\[ \text{main}(d_{1},\ldots,d_{k}) \rightarrow_{A_{i}} C[\sigma] \rightarrow_{A} C[r_{s} \sigma], \]

and thus \((\sigma, r_{s} \sigma) \in Z_{i}\). By assumption, there exists a substitution \( \sigma_{i,j} (i \in \{1,\ldots,k\}) \) is at least as general as \( \sigma \). Hence the ATRS \( B \) can simulate the step from \( C[\sigma] \rightarrow_{A} C[r_{s} \sigma] \), using the rule \( l_{i} \rightarrow_{A} r_{s} \sigma \). From this, the property follows by inductive reasoning.

As a consequence, the collecting semantics of \( A \) 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 \((Z_{1},\ldots,Z_{n})\) 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 \( G \) can be seen as a ground ATRS whose left-hand sides are all function symbols with arity zero. The non-terminals of \( G \) are precisely the left-hand sides. For the remaining, we assume that variables occurring \( A \) are indexed by indices of rules, i.e. every variable occurring in the \( i^{th} \) rule \( l_{i} \rightarrow r_{i} \in A \) has index \( i \). Hence the set of variables of rewrite rules in \( A \) are pairwise disjoint. Besides a designated non-terminal \( S \), the start-symbol, the constructed tree grammar \( G \) admits two kinds of non-terminals: non-terminals \( R_{i} \) for each rule \( l_{i} \rightarrow r_{i} \) and non-terminals \( z_{i} \) for variables \( z_{i} \) occurring in \( A \). Note that the variable \( z_{i} \) is considered as a constant in \( G \). We say that \( G \) is safe for \( A \) if the following two conditions are satisfied for all \((\sigma, l) \in Z_{i}: (i) z_{i} \rightarrow_{G} \sigma(z_{i}) \) for each \( z_{i} \in \text{Var}(l_{i}); \) and (ii) \( R_{i} \rightarrow_{G} t \). This way, \( G \) constitutes a finite over-approximation of the collecting semantics of \( A \).

**Example 11.** Figure 2 shows the tree grammar \( G \) constructed by the method described below, which is safe for the ATRS \( A \) from Example 6. The notation \( N \rightarrow t_{1} \mid \cdots \mid t_{n} \) is a short-hand for the \( n \) rules \( N \rightarrow t_{i} \).

The construction of Jones consists of an initial automaton \( G_{0} \), which describes considered start terms, and which is then systematically closed under rewriting by way of an extension
We let the tree grammar $G$ be the least set of rules satisfying $G \supseteq G_0 \cup \delta(G)$ with
\[
\delta(G) := \bigcup_{N \rightarrow C[u] \in G} \text{Ext}^\text{cbv}(N \rightarrow C[u]).
\]
Here, $\text{Ext}^\text{cbv}(N \rightarrow C[u])$ is defined as the following set of rules:
\[
\begin{align*}
N &\rightarrow C[R_i], \quad l_i \rightarrow r_i, \quad \sigma_i \in \text{Var}, \\
R_i &\rightarrow \gamma_i, \quad \text{and} \\
z_i \rightarrow \sigma(z_i) \quad \text{for all } z_i \in \text{Var}(l_i) \quad \text{and} \quad \sigma \text{ normalised wrt. } A.
\end{align*}
\]
In contrast to [33], we require that the substitution $\sigma$ is normalised, thereby modelling call-by-value semantics. The tree grammar $G$ is computable using a simple fix-point construction. Minimality of $f(t_1, \ldots, t_k) \rightarrow^* \ell_0, l_0$ means that there is no shorter sequence $f(t_1, \ldots, t_k) \rightarrow^* l_0, \tau$ with $l_0, \tau \rightarrow^* l_0, \sigma$, and ensures that $G$ 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 $G_0$ given by the rule
\[
S \rightarrow \text{main}(\ast) \quad \ast \rightarrow \emptyset \mid \ast ; \ast ,
\]
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 main($\ast$) matches the (renamed) rule $\{12\}$ main($\emptyset$) $\rightarrow$ fixwalk$\ast$. Using the substitution $\{12\} \rightarrow *$, we fix the violation by adding productions
\[
S \rightarrow R_{12}, \quad R_{12} \rightarrow \text{fixwalk}(\emptyset) \ast, \quad R_{12} \rightarrow \ast.
\]

The tree grammar $G$ constructed so far tells us that $\{12\}$ 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 fixwalk (rules $[9]$ and $[10]$):
\[
\begin{align*}
\text{fixwalk} &\rightarrow R_{12} \rightarrow \ast, \\
\text{fixwalk} &\rightarrow R_{12} \rightarrow \ast ; \ast.
\end{align*}
\]

To resolve the closure violations, the tree grammar is extended by productions
\[
R_{12} \rightarrow R_{12} \cdot \ast, \quad R_{12} \rightarrow \ast \cdot R_{12}
\]
because of rule $[9]$, and by
\[
R_{12} \rightarrow R_{12} \cdot \ast, \quad R_{12} \rightarrow \ast \cdot R_{12} \ast \rightarrow \ast, \\
| C_{12}(R_{12}, C_{12}(\ast)) \ast \rightarrow \ast, \quad C_{12}(R_{12}, C_{12}(\ast)) \ast \rightarrow \ast.
\]
due to rule $[10]$. We can now identify a new violation in the production of $R_{12}$. Fixing all violations this way will finally result in the tree grammar depicted in Figure 2.

The following lemma confirms that $G$ 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 $S \rightarrow \emptyset t$ and $t \rightarrow_{A} C[l, \sigma] \rightarrow_{A} C[r, \sigma]$ then $S \rightarrow_{G} C[R_i], R_i \rightarrow_{A} r_i$, and $z_i \rightarrow_{G} \sigma(z_i)$ for all $z_i \in \text{Var}(l_i)$.

**Theorem 4.** The tree grammar $G$ is safe for $A$.

Proof. Fix $(\sigma, t) \in Z_i$, and let $z \in \text{Var}(l_i)$. Thus main($\emptyset$) $\rightarrow_{A} C[l, \sigma] \rightarrow_{A} C[r, \sigma]$ and $r_i \rightarrow_{A} t$ for some inputs $d_i \in \text{Inp}$. As we have $S \rightarrow_{G} \text{main}(d) \subseteq G$, Lemma 2 yields $R_i \rightarrow_{A} r_i$, and $z_i \rightarrow_{G} \sigma(z_i)$. The second safeness conditions is satisfied. Clearly, $R_i \rightarrow_{A} r_i \rightarrow_{A} r_i, r_i, s$. A standard induction on the length of $r_i, \rightarrow_{A} t$ then yields $R_i \rightarrow_{G} t$, using again Lemma 2.

We arrive now at our conrete implementation $\text{cfa}(A)$ that employs the above outlined call flow analysis to deal with both dead code elimination and instantiation on the given ATRS $A$. The construction of the tree grammar $G$ itself closely follows the algorithm outlined by Jones [33]. Recall that the $3^{rd}$ rule $l_i \rightarrow r_i \in A$ constitutes dead code if the $3^{rd}$ component $Z_i$ of the collecting semantics of $A$ is empty, by Lemma 2. Based on the constructed tree grammar, the implementation identifies rule $l_i \rightarrow r_i$, as dead code when $G$ does not define a production $R_i \rightarrow t$, and thus $Z_i = \emptyset$. All such rules are eliminated, in accordance to Proposition 4. On the remaining rules, our implementation performs instantiation as follows. We suppose $\pi$-productions $N \rightarrow M$, for non-terms $M$, have been eliminated by way of a standard construction, preserving the set of terms from non-terms in $G$. Thus productions in $G$ have the form $N \rightarrow (t_1, \ldots, t_k)$. Fix a rule $l_i \rightarrow r_i \in A$. The primary goal of this stage is to get rid of head variables, with respect to the $3^{rd}$-saturated ATRS $A_{3^{rd}}$, thereby enabling uncurrying so that the ATRS $A$ can be brought into functional form. For all such head variables $z$, then, we construct a set of binders
\[
\{z_i \mapsto \text{fresh}(f(t_1, \ldots, t_k)) \mid z_i \rightarrow f(t_1, \ldots, t_k) \in G\},
\]
where the function fresh replaces non-terms by fresh variables, discarding binders where the right-hand contains defined variables. For variables $z$ which do not occur in head positions, we construct such a binder only if the production $z_i \rightarrow f(t_1, \ldots, t_k)$ is unique. With respect to the tree grammar of Figure 2, the implementation generates binders
\[
\{f_1 \mapsto C_{12}, f_1 \mapsto C_{12}(f_1', C_{12}(x'))\} \cup \{g_1 \mapsto C_{12}(x')\}.
\]

The product-combination of all such binders gives then a set of substitution $\{\sigma_1, \ldots, \sigma_k\}$ that leads to sufficiently many instantiations $l_i, \sigma_1, \ldots, r_i, \sigma_k$ of rule $l_i \rightarrow r_i$, by Lemma 2. Our implementation replaces every rule $l_i \rightarrow r_i \in A$ 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...
We have now seen all the building blocks underlying our tool ATRSs to functional form by the uncurrying transformation, using expensive version based on control flow analysis in case the simplifies the obtained TRS by controlled inlining, applying syntax further eliminate dead code. The transformation control flow analysis to instantiate head variables sufficiently and toTRS defunctionalisation, followed by syntax driven dead code elimination based, criterion from [24] to eliminate dead code (see Sections 5.1 and 5.2, respectively, the transformation itself defined from transformations out of three transformations, viz, complexity reflection and 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 $f_2 \circ f_1$, where $f_1 \circ f_2(A) = f_1(f_2(A))$ if defined and $f_1 \circ f_2(A) = A$ otherwise;
- the transformation exhaustive $f$ iterates the transformation $f$ until inapplicable on the current ATRS; and
- the operator $\triangleright$ implements left-handed choice: $f_1 \triangleright f_2$ 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 4.3.

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 inline(usableRules \triangleright cfaDCE) of l, 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 TTT2 [5], version 2.1. For comparison, we have also paired HOCA with the termination tool TTT2 [5], version 1.15.

In Table 1 we summarise our experimental findings on the 25 examples from our collection. Row 5 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 TTT2, and those programs that can be shown terminating when HOCA is paired with TTT2. In contrast, row 0 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. 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 automated 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

We ran also experiments with AProVE and Omega 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 2851http://www.cpubenchmark.net/
What this paper shows is that complexity analysis of higher-order
which cannot be handled by linear type systems [13]: we do
collection semantics to complexity analysis is novel.
function which in an higher-order variable occurs free. Also, type systems like the ones in [35], which are restricted to
control flow analysis, our addition of
one 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. Consequentially, the body of literature on control flow analysis is considerable, see e.g. the recent survey of Midtgaard [59]. 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 [54] 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 [50] 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 enforce 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 intentionally 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 full 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 [20] 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

### Table 1. Experimental Evaluation conducted with $\mathcal{T}$ and $\mathcal{P}_2$.

<table>
<thead>
<tr>
<th></th>
<th>constant</th>
<th>linear</th>
<th>quadratic</th>
<th>polynomial</th>
<th>terminating</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td># systems</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>FOP execution time</td>
<td>0.37/1.71/3.05</td>
<td>0.37/4.82/13.85</td>
<td>0.37/4.82/13.85</td>
<td>0.37/4.82/13.85</td>
</tr>
<tr>
<td>S</td>
<td># systems</td>
<td>2</td>
<td>14</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>HDTCA execution time</td>
<td>0.01/2.28/4.56</td>
<td>0.01/0.54/4.56</td>
<td>0.01/0.43/4.56</td>
<td>0.01/0.42/4.56</td>
</tr>
<tr>
<td></td>
<td>FOP execution time</td>
<td>0.23/0.51/0.79</td>
<td>0.23/2.53/14.00</td>
<td>0.23/6.30/30.12</td>
<td>0.23/10.94/60.10</td>
</tr>
</tbody>
</table>

isort-fold.fp, which is parametric on the subroutine which compares the elements of the list being sorted. This is an example 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. HDTCA, instead, is able to put it in a form which allows $\mathcal{T}$ 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. HDTCA 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.
programs related to these restrictions, and only works with primitive recursive definitions.

References


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


