Higher-Order Complexity Analysis: Harnessing First-Order Tools.

Martin Avanzini[b], Ugo Dal Lago[b], Georg Moser[a]

[a] *University of Innsbruck, Austria*
[b] *Università di Bologna & INRIA, Sophia Antipolis, Italy*

Keywords: Functional Programming, Defunctionalisation, Term Rewriting, Termination and Resource Analysis

1. Introduction

Termination, and more broadly *resource analysis*, is important to a number of areas, including embedded real-time and safety-critical systems. Programs written in a purely functional language, where side effects are discouraged, are particular well-suited to such a static analysis. Over the last decades, there has been significant work on the analysis of first-order functional programs. Notably, the rewriting-community has developed several tools for the automatic termination and resource analysis of *term rewrite system*, a formal model of computation that is at the heart of functional programs, e.g. the termination provers `AProVE` [5] and $\mathsf{T_TT_2}$ [9], and our own complexity-analyser $\mathsf{T_CT}$ [2].

In this note we investigate to which extend such tools can be put into use for the analysis of *higher-order*, functional programs. To this end, we essentially rely on *defunctionalisation* [12], a whole-program transformation from higher-order to first-order functional programs. The idea behind this transformation is conceptually simple: function-abstracts 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 rev l =
  (* walk : 'a list → ('a list → 'a list) *)
  let rec walk = function
       [] → (fun ys → ys)
     | x :: xs → comp (walk xs) (fun ys → x :: ys)
  in walk l [];;
```

Run on a list of n elements, the function `rev` first constructs n nested closures, which when further evaluated, results in the input list reversed. Defunctionalisation of this program consists in first defining a datatype for representing the three abstractions **fun** `z` → f (g `z`), **fun** ys → ys and **fun** ys → x :: ys respectively:

```
type 'a closure = Lam1 of ('a closure, 'a closure) | Lam2 | Lam3 of 'a ;;
```

More precise, an expression of type `'a closure` represents a function closure, arguments are used to store assignments to free variables. An appropriate apply-function can then be defined as follows.

```
let rec apply c a = match c with
     Lam1(f,g) → apply f (apply g a) | Lam2 → a | Lam3(x) → x :: a ;;
```

☆This work was partially supported by FWF project number J3563, FWF project number P25781 and by French ANR project Elica ANR-14-CE25-0005.

Email addresses: `martin.avanzini@uibk.ac.at` (Martin Avanzini), `dallago@cs.unibo.it` (Ugo Dal Lago), `georg.moser@uibk.ac.at` (Georg Moser)

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

```
let comp f g = Lam1(f,g);;
let rev l =
  (* walk : 'a list → 'a list closure *)
  let rec walk = function
        [] → Lam2
     | x :: xs → comp (walk xs) Lam3(x)
  in apply (walk l) [];;
```

Observe that now the recursive function `walk` constructs an explicit representation of the closure computed by its original definition. The function `apply` carries out the remaining evaluation. This definition can now be easily turned into a first-order term rewrite system.

To evaluate defunctionalisation in the context of static analysis with term rewriting tools, we have build `HOCA`.[1] This tool is able to translate programs written in the pure subset of `OCaml`, as e.g. the example from above, into a term rewrite system. Crucially, the transformation steps carried out by our tool are all step preserving, modulo a constant factor. Thus a successful termination, or runtime-complexity analysis, on the resulting rewrite system can be relayed back to the initial `OCaml`-program. In the remaining, we outline the theory underlying `HOCA`, specifically:

1. In Section 2 we present a transformation from higher-order programs to *applicative term rewrite systems* (*ATRSs* for short). This program-transformation is a variation of the one found in [4] for λ-calculus endowed with a weak call-by-value reduction strategy, and essentially an instance of defunctionalisation.
2. To faithfully model the evaluation of `OCaml`-programs, we have to take care of partial applications during defunctionalisation. As a consequence, the transformation to ATRSs ends up in a system that can be seen as an interpreter specialized to the input program, with a single recursive apply-function. Lacking sufficient structure, all tools outlined above perform poor on the resulting systems. To overcome this situation, we propose in Section 3 several complexity-preserving transformations on ATRSs.
3. We have compiled so far a small testbed of higher-order programs, including 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 a small parsers relying on the monadic parser-combinator outlined in Okasaki's functional pearl [10].

2. Complexity Preserving Transformation

In the following, we assume modest familiarity with term rewriting [3] and functional programming [6]. To model a reasonable rich but pure subset of `OCaml`, we consider Plotkin's *Programming Computable Functions* (*PCF* for short) [11]. To seamlessly express programs over algebraic datatypes, we enrich Plotkin's original definition with constructors and pattern matching. To this end, let $\mathbf{c}_1, \ldots, \mathbf{c}_k$ be finitely many constructors, each equipped with a fixed *arity*. The syntax of PCF-programs is given by the following grammar:

$$\mathsf{Exp} \quad e, f \ ::= \ x \mid \mathbf{c}_i(\vec{e}) \mid \lambda x.e \mid f\ e \mid \mathsf{fix}(x.e) \mid \mathsf{match}\ e\ \mathsf{with}\ \{\mathbf{c}_1(\vec{x}_1) \mapsto e_1; \cdots; \mathbf{c}_k(\vec{x}_k) \mapsto e_k\}\ ,$$

where x ranges over variables. We adopt *weak call-by-value* semantics, the definition is standard, see e.g. [6]. The one-step weak call-by-value reduction relation is denoted by $\to_v$. Elements of the term algebra over constructors $\mathbf{c}_1, \ldots, \mathbf{c}_k$ embedded in our language are collected in Data. A *PCF-program* with n *input arguments* x_i is a closed expression $P = \lambda x_1 \cdots \lambda x_n.e$. We only considered elements of Data as valid input to a program, in particular, inputs are free of abstractions. This way we 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. Therefore, the free variables $\mathsf{FV}(e)$ of e can be defined as an ordered sequence of variables. Note that the variables $\vec{x}_i$ in a match-expression are considered bound in e_i.

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

In the following, we show that every PCF-program P can be seen as an *applicative term rewrite system* (*ATRS* for short) $\mathcal{R}_P$, a system with a dedicated infix symbol $(\cdot)$, the *apply-symbol.* We define a mapping $[\cdot]_\Phi$ from expressions to terms as follows. In the definition of $[e]_\Phi$ we abbreviate $\vec{x} = \mathsf{FV}(e)$.

$$\begin{array}{lll}[x]_\Phi := x & [\lambda x.e]_\Phi := \mathtt{lam}[\mathsf{x.e}](\vec{x}) & [\mathbf{c}_i(e_1,\dots,e_k)]_\Phi := \mathbf{c}_i([e_1]_\Phi,\dots,[e_k]_\Phi) \\ [f\ e]_\Phi := [e]_\Phi \cdot [f]_\Phi & [\mathsf{fix}(x.e)]_\Phi := \mathtt{fix}[\mathsf{x.e}](\vec{x}) & [\mathsf{match}\ e\ \mathsf{with}\ \{cs\}]_\Phi := \mathtt{match}[\mathsf{e,cs}]([e]_\Phi,\vec{x})\ .\end{array}$$

Each of the introduced function symbols $\mathtt{lam}[\mathsf{x.e}]$, $\mathtt{fix}[\mathsf{x.e}]$, and $\mathtt{match}[\mathsf{e,cs}]$, called *closure-constructors* below, is equipped with one or more *defining rules*:

- $\mathtt{lam}[\mathsf{x.e}]$ is defined by $\mathtt{lam}[\mathsf{x.e}](\vec{x}) \cdot x \to [e]_\Phi$;
- $\mathtt{fix}[\mathsf{x.e}]$ is defined by $\mathtt{fix}[\mathsf{x.e}](\vec{x}) \cdot y \to [f]_\Phi \cdot y$ for y a fresh variable and $f = e\{\mathsf{fix}(x.e)/x\}$;
- $\mathtt{match}[\mathsf{e,cs}]$, where $cs = \mathbf{c}_1(\vec{x}_1) \mapsto e_1; \cdots ; \mathbf{c}_k(\vec{x}_k) \mapsto e_k$, is defined by $\mathtt{match}[\mathsf{e,cs}](\mathbf{c}_i(\vec{x}_i),\vec{x}) \to [e_i]_\Phi$ for $1 \leqslant i \leqslant k$.

For a program $P = \lambda x_1 \cdots \lambda x_n.e$, the ATRS $\mathcal{R}_P$ is defined as the least set of rules that (i) contains a rule $\mathtt{main}(x_1,\dots,x_n) \to [e]_\Phi$; and (ii) if $l \to r \in \mathcal{R}_P$ and $\mathtt{f}$ is a closure-constructor in r, it contains all defining rules of $\mathtt{f}$. Note that the size of $\mathcal{R}_P$ is bounded by the number of distinct sub-expressions of e.

Example 1. *The example from the introduction is translated into the following ATRS. Here* $\mathtt{walk}$ *is used to abbreviate the λ-expression that constitutes the body of* $\mathtt{walk}$*, similar, abbreviations are used for the two cases in* $\mathtt{walk}$ *and the sub-expressions of* $\mathtt{comp}$*.*

$$\begin{array}{lrl}
(a) & \mathtt{main}(l) \to \mathtt{fix}[\mathtt{walk}] \cdot l \cdot \mathbf{nil} & (e)\quad \mathtt{lam2} \cdot ys \to ys \\
(b) & \mathtt{fix}[\mathtt{walk}] \cdot l \to \mathtt{walk} \cdot l & (f)\quad \mathtt{lam3}(x) \cdot ys \to \mathbf{cons}(x,ys) \\
(c) & \mathtt{walk} \cdot \mathbf{nil} \to \mathtt{lam2} & (g)\quad \mathtt{lam1}_0 \cdot f \to \mathtt{lam1}_1(f) \\
(d) & \mathtt{walk} \cdot \mathbf{cons}(x,xs) \to \mathtt{lam1}_0 \cdot (\mathtt{fix}[\mathtt{walk}] \cdot xs) \cdot \mathtt{lam3}(x) & (h)\quad \mathtt{lam1}_1(f) \cdot g \to \mathtt{lam1}_2(f,g) \\
 & & (i)\quad \mathtt{lam1}_2(f,g) \cdot z \to f \cdot (g \cdot z)
\end{array}$$

Proposition 1. *Every $\to_v$-reduction of an expression $P\ d_1\ \cdots\ d_n$ ($d_j \in \mathsf{Data}$) is simulated step-wise by an innermost $\mathcal{R}_P$-derivation starting from* $\mathtt{main}(d_1,\dots,d_n)$*.*

Here innermost means that rewriting is performed inside-out. As the inverse direction of this proposition can also be stated, $\mathcal{R}_P$ can be seen as a sound and complete, in particular step-preserving, implementation of the PCF-program P. This simple transformation paves the way to our primary goal. A successful analysis of $\mathcal{R}_P$ with rewriting-based termination and runtime-complexity tools can be relayed back to the PCF-program P.

3. Complexity-Preserving Simplifications

Rule-Narrowing. The purpose of our first transformation is to simplify rules by rewriting right-hand sides. For instance, the two applications in rule (h) of Example 1 could already be performed directly on the rule. Such a simplification can be seen as a form of *inlining.* In a similar way, we can inline the calls to $\mathtt{walk}$ in rule (b) by first instantiating the variable l to $\mathbf{nil}$ and $\mathbf{cons}(x,xs)$, respectively. This process can be formalized by *narrowing* [3], consequently we call this transformation *rule-narrowing.* It involves replacing a rewrite rule of the form $l \to r[\mathtt{f}(\vec{r})]$, where the symbol $\mathtt{f}$ is possibly the infix apply-symbol, by the set of rules

$$\{l\mu_i \to r\mu_i[r_i\mu_i] \mid \text{the sub-term } \mathtt{f}(\vec{r}) \text{ unifies with the left-hand side of a rule } \mathtt{f}(\vec{l_i}) \to r_i \text{ with mgu } \mu_i\}\ .$$

Here it is supposed that variables are renamed apart during unification. The unifiable rules $\mathtt{f}(\vec{l_i}) \to r_i$ are called *narrowing-rules* for brevity. Notice that the rule $l\mu_i \to r\mu_i[r_i\mu_i]$ corresponds to the specialization of $l \to r[\mathtt{f}(\vec{r})]$ by μ_i, with $\mathtt{f}$ inlined.

Example 2 (Continued from Example 1). *We can narrow rule (b) with narrowing-rules (c) and (d), using the unifiers* $\{l \mapsto \mathbf{nil}\}$ *and* $\{l \mapsto \mathbf{cons}(x, xs)\}$*, resulting in the rules*

$$\mathtt{fix[walk]} \cdot \mathbf{nil} \to \mathtt{lam2} \qquad \mathtt{fix[walk]} \cdot \mathbf{cons}(x, xs) \to \mathtt{lam1}_0 \cdot (\mathtt{fix[walk]} \cdot xs) \cdot \mathtt{lam3}(x) \ .$$

Applying rule-narrowing on the second rule twice, we obtain the system consisting of the following six rules.

$$\begin{array}{llll}
(1) & \mathtt{main}(l) \to \mathtt{fix[walk]} \cdot l \cdot \mathbf{nil} & (4) & \mathtt{lam2} \cdot ys \to ys \\
(2) & \mathtt{fix[walk]} \cdot \mathbf{nil} \to \mathtt{lam2} & (5) & \mathtt{lam3}(x) \cdot ys \to \mathbf{cons}(x, ys) \\
(3) & \mathtt{fix[walk]} \cdot \mathbf{cons}(x, xs) \to \mathtt{lam1}_2(\mathtt{fix[walk]} \cdot xs, \mathtt{lam3}(x)) & (6) & \mathtt{lam1}_2(f, g) \cdot z \to f \cdot (g \cdot z) \ .
\end{array}$$

Notice that the inlined rules (c),(d),(g) and (h), which are no longer used in a reduction of `main`*, have been dropped* [2].

Rule-narrowing reduces the number of reduction steps, but crucially, only by a constant factor if some care is taken. Under the following pre-conditions, a successful analysis on the transformed system can be relayed back to the input system.

First, it has to be guaranteed that a redex with respect to a replaced rule $l \to r[\mathtt{f}(\vec{r})]$, reachable from a call to `main`, is a redex to a generated rule $l\mu_i \to r\mu_i[r_i\mu_i]$. On non-overlapping systems such as the ones we generate, this can be guaranteed by requiring that the symbol `f` is *sufficiently defined*: any sub-term $\mathtt{f}(\vec{t})$ occurring in a reduction in any reduction starting from $\mathtt{main}(d_1, \dots, d_n)$ ($d_j \in \mathsf{Data}$) is reducible. Secondly, it has to be ensured that inlining does not delete redexes. A sufficient condition for this purpose is to require that for each narrowing-rule $\mathtt{f}(\vec{l_i}) \to r_i$, each variable occurs in the right-hand side r_i at least as often as it occurs in the left-hand side $\mathtt{f}(\vec{l_i})$. We call such rules *redex-preserving.* Then the following can be shown.

Proposition 2. *Let* $\mathcal{S}$ *be obtained from the non-overlapping TRS* $\mathcal{R}$ *by narrowing the rule* $l \to r[\mathtt{f}(\vec{r})]$ *as outline above. If* `f` *is sufficiently defined and all narrowing-rules are redex-preserving, then every derivation wrt.* $\mathcal{R}$ *starting from* $\mathtt{main}(d_1, \dots, d_n)$ $(d_j \in \mathsf{Data})$ *of length* ℓ *is simulated by a derivation wrt.* $\mathcal{S}$ *from* $\mathtt{main}(d_1, \dots, d_n)$ *of length* $\lfloor \frac{\ell}{2} \rfloor$.

Instantiation. Usually the *dependency pair transformation*, a form of call-graph analysis, is performed as a first step by termination or complexity analysers (see [5, 9, 2]). This method is ineffective on the applicative systems we generate. To overcome this situation, we intend to transform applicative terms $\mathtt{f}(t_1, \dots, t_m) \cdot t_{m+1} \cdots t_{m+n}$ to *functional form* $\mathtt{f}_n(t_1, \dots, t_m, t_{m+1}, \dots, t_{m+n})$. It is well understood how to bring an ATRS into functional form, in a step preserving manner [7]. In the presence of *head-variables*, i.e. variables that occur to the left of the apply-symbol such as in rule (*6*), the apply-symbol cannot be eliminated.

To overcome this situation, we instantiate such variables by closure-terms. Call an ATRS $\mathcal{S}$ a *refinement* of an ATRS $\mathcal{R}$ if every rule $l' \to r' \in \mathcal{S}$ is an instance of a rule $l \to r \in \mathcal{R}$, i.e., $l' = l\sigma$ and $r = r'\sigma$ for some substitution σ. It is a *safe refinement* if for every term t occurring in a derivation wrt. $\mathcal{R}$ from $\mathtt{main}(d_1, \dots, d_n)$ ($d_j \in \mathsf{Data}$), every redex in t wrt. $\mathcal{R}$ is also a redex wrt. $\mathcal{S}$. The following proposition then follows by definition.

Proposition 3. *If* $\mathcal{S}$ *is a safe refinement of* $\mathcal{R}$*, then every derivation wrt.* $\mathcal{R}$ *starting from* $\mathtt{main}(d_1, \dots, d_n)$ $(d_j \in \mathsf{Data})$ *is simulated step-wise by an derivation wrt.* $\mathcal{S}$*, and vice verse.*

The question now remains how to construct a suitable, safe refinement of the ATRS under consideration. To this end, we use a variation of the flow-analysis of Jones [8] tailored to innermost reductions. Jones defines an over-approximation of the *collecting semantics*, using regular tree grammars. An initial grammar, which expresses analysed function calls, is closed under rewriting in a systematic way. In particular, each variable x occurring in the i^{th} rule is represented as a non-terminal X_i in the closed grammar $\mathcal{G}$, the set of terms generated from the non-terminal X_i approximates the assignments to x.

[2] Our tool `HOCA` reduces the rewrite system automatically to the set of *usable* rules, an idea that goes back to [1].

$$\begin{aligned}
\star &\to \mathbf{nil} \mid \mathbf{cons}(\star,\star) & R_1 &\to \mathtt{fix[walk]} \cdot L_6 \cdot \mathbf{nil} \mid R_2 \cdot \mathbf{nil} \mid R_3 \cdot \mathbf{nil} \mid R_4 \mid R_6 \\
\mathsf{St} &\to \mathtt{main}(\star) & R_2 &\to YS_4 \\
L_1, X_3, XS_3, X_5, Z_6 &\to \star & R_3 &\to \mathtt{lam1}_2(\mathtt{fix[walk]} \cdot XS_3, \mathtt{lam3}(X_3)) \\
YS_4, Z_6 &\to R_4 \mid \mathbf{nil} & &\quad \mid \mathtt{lam1}_2(R_2, \mathtt{lam3}(X_3)) \mid \mathtt{lam1}_2(R_3, \mathtt{lam3}(X_3)) \\
YS_5 &\to Z_6 & R_4 &\to \mathtt{lam2} \\
F_6 &\to R_2 \mid R_3 & R_5 &\to \mathbf{cons}(X_5, YS_5) \\
G_6 &\to \mathtt{lam3}(X_3) & R_6 &\to F_6 \cdot (G_6 \cdot Z_6) \mid F_6 \cdot R_5 \mid R_4 \mid R_6
\end{aligned}$$

Figure 1: Approximation of the Collecting Semantics by a Regular Tree Grammar.

Example 3 (Continued from Example 2). *We refine rule (6) to the two head-variable free rules, using rule-narrowing to simplify the right-hand sides:*

$$\begin{aligned}
&\textit{(6a)} & \mathtt{lam1}_2(\mathtt{lam2}, \mathtt{lam3}(x)) \cdot z &\to \mathbf{cons}(x, z) \\
&\textit{(6b)} & \mathtt{lam1}_2(\mathtt{lam1}_2(f, \mathtt{lam3}(y)), \mathtt{lam3}(x)) \cdot z &\to \mathtt{lam1}_2(f, \mathtt{lam3}(y)) \cdot \mathbf{cons}(x, z)\ .
\end{aligned}$$

The regular tree grammar depicted in Figure 1 is obtained from Jones flow analysis. From the productions of F_6 and G_6 it can be seen that assignments to the variables f and g of rule (6) are of the form `lam2` *or* $\mathtt{lam1}_2(f, \mathtt{lam3}(y))$ *and* $\mathtt{lam3}(x)$*, respectively. This confirms that our refinement is sound.*

4. Experimental Results

We have outlined a transformation from higher-order programs to first-order term rewrite systems, using standard program analysis methods and rewriting-based simplification techniques. In Table 1 we summarise our experimental findings on 21 examples, detailed results can be found on the `HOCA` homepage. As back-ends we use `AProVE` and TTT2 for termination, and TCT for runtime-complexity analysis. The rows indicate the derived asymptotic upper-bounds, SN denotes that the system could be proven terminating. The first column shows the effectiveness of the tools on the applicative term rewrite systems obtained by Proposition 1, in the second column we integrate the techniques of Section 3 and *uncurrying* [7].

	P. 1	P. 1, 2 & 3
$O(1)$	1	1
$O(n)$	5	12
$O(n^2)$	5	16
SN	11	20
Failed	10	1

Table 1: Experimental Evaluation conducted with `AProVE`, TCT and TTT2.

These preliminary experiment confirm at what we already hinted in the introduction. Existing rewriting-based static analysis tools perform relatively poor on defunctionalized programs, but suitable simplifications can massage the analysed ATRS into a form amendable for automatic analysis. Noteworthy, this way we could show all but one example from our test-suite terminating.

References

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

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

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

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

[5] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proc. of 3rd IJCAR*, volume 4130 of *LNAI*, pages 281–286. Springer, 2006.

[6] R. Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, 2012.

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

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

[9] 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. Springer, 2009.

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

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

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