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

Martin Avanzini\textsuperscript{b}, Ugo Dal Lago\textsuperscript{b}, Georg Moser\textsuperscript{a}

\textsuperscript{a}University of Innsbruck, Austria
\textsuperscript{b}Universit\`a 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\cite{5} and TTT2\cite{9}, and our own complexity-analysers TCT\cite{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\cite{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 \texttt{OCaml}-program:

\begin{verbatim}
let comp f g = fun z \rightarrow f (g z); ;
let rec rev l =
  (* walk : 'a list \rightarrow ('a list \rightarrow 'a list) *)
  let rec walk = function
    [] \rightarrow (fun ys \rightarrow ys)
    | x :: xs \rightarrow comp (walk xs) (fun ys \rightarrow x :: ys)
  in walk l [] ; ;
\end{verbatim}

Run on a list of n elements, the function \texttt{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 \texttt{fun z \rightarrow f (g z)}, \texttt{fun ys \rightarrow ys} and \texttt{fun ys \rightarrow z :: ys} respectively:

\begin{verbatim}
  type 'a closure = Lam1 of ('a closure , 'a closure ) | Lam2 | Lam3 of 'a ; ;
\end{verbatim}

More precise, an expression of type \texttt{'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.

\begin{verbatim}
let rec apply c a = match c with
  Lam1(f,g) \rightarrow apply f (apply g a) | Lam2 \rightarrow a | Lam3(z) \rightarrow z :: a ; ;
\end{verbatim}

\textsuperscript{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:

```ocaml
let comp f g = Lam1(f, g);;
let rec walk =
  (* walk : 'a list → 'a list closure *)
  function [] → Lam2
  | z :: zs → comp (walk zs) Lam3(z)
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 \[^2\] 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 \[^10\].

\[
\begin{align*}
\text{Exp} &\ ::= & x & | & c_i(e) & | & \lambda x.e & | & f e & | & \text{fix}(x.e) & | & \text{match } e \text{ with } \{c_1(x_1) \mapsto e_1; \cdots ; c_k(x_k) \mapsto e_k\},
\end{align*}
\]

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 \(\rightarrow_v\). Elements of the term algebra over constructors \(c_1, \ldots , c_k\) 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 \(\text{FV}(e)\) 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 \url{http://cbruibk.ac.at/tools/hoca/}\.
In the following, we show that every PCF-program \( P \) can be seen as an applicative term rewrite system \( (\text{ATRS}) \) \( \mathcal{R}_P \), a system with a dedicated infix symbol \((\cdot)\), the apply-symbol. We define a mapping \([e]_\Phi\) from expressions to terms as follows. In the definition of \([e]_\Phi\) we abbreviate \( \tilde{x} = \text{FV}(e) \).

\[
\begin{align*}
[x]_\Phi & := x \\
[\lambda x.e]_\Phi & := \text{lam}[x.e](\tilde{x}) \quad [c_1(e_1, \ldots, e_k)]_\Phi & := c_1([e_1]_\Phi, \ldots, [e_k]_\Phi) \\
[f e]_\Phi & := [e]_\Phi \cdot [f]_\Phi \quad [\text{fix}(x.e)]_\Phi & := \text{fix}[x.e](\tilde{x}) \quad [\text{match } e \text{ with } \{cs\}]_\Phi & := \text{match}[e, cs][[e]_\Phi, \tilde{x}].
\end{align*}
\]

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

- \text{lam}[x.e] is defined by \text{lam}[x.e](\tilde{x}) \cdot x \rightarrow [e]_\Phi;
- \text{fix}[x.e] is defined by \text{fix}[x.e](\tilde{x}) \cdot y \rightarrow [f]_\Phi \cdot y \text{ for } y \text{ a fresh variable and } f = e \text{fix}[x.e]/x;
- \text{match}[e, cs], where \( cs = c_1(\tilde{x}_1) \mapsto e_1; \ldots; c_k(\tilde{x}_k) \mapsto e_k, \) is defined by \text{match}[e, cs][c_1(\tilde{x}), \tilde{x}] \rightarrow [e_i]_\Phi \text{ for } 1 \leq i \leq k.

For a program \( P = \lambda x_1 \ldots x_n. e, \) the ATRS \( \mathcal{R}_P \) is defined as the least set of rules that (i) contains a rule \( \text{main}(x_1, \ldots, x_n) \rightarrow [e]_\Phi; \) and (ii) if \( l \rightarrow r \in \mathcal{R}_P \) and \( f \) is a closure-constructor in \( r, \) it contains all defining rules of \( f. \) 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 \text{walk} is used to abbreviate the \( \lambda \)-expression that constitutes the body of \text{walk}, similar, abbreviations are used for the two cases in \text{walk} and the sub-expressions of \( \text{comp}. \)

\[
\begin{align*}
(a) & \quad \text{main}(l) \rightarrow \text{fix}[\text{walk}] \cdot l \cdot \text{nil} & \quad (e) & \quad \text{lam}_2: y s \rightarrow y s \\
(b) & \quad \text{fix}[\text{walk}] \cdot l \rightarrow \text{walk} \cdot l & \quad (f) & \quad \text{lam}_3(x) \cdot y s \rightarrow \text{cons}(x, y s) \\
(c) & \quad \text{walk} \cdot \text{nil} \rightarrow \text{lam}_2 & \quad (g) & \quad \text{lam}_1_0: f \rightarrow \text{lam}_1(f) \\
(d) & \quad \text{walk} \cdot \text{cons}(x, x s) \rightarrow \text{lam}_1_0 \cdot (\text{fix}[\text{walk}] \cdot x s) \cdot \text{lam}_3(x) \\
(h) & \quad \text{lam}_1_1(f) \cdot g \rightarrow \text{lam}_1_2(f, g) \\
(i) & \quad \text{lam}_1_2(f, g) \cdot z \rightarrow f \cdot (g \cdot z)
\end{align*}
\]

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

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 \text{walk} in rule \((b)\) by first instantiating the variable \( l \) to \text{nil} and \text{cons}(x, x s), 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 \( l \rightarrow r[f(\tilde{r})], \) where the symbol \( f \) is possibly the infix apply-symbol, by the set of rules

\[
\{l\mu_i \rightarrow r\mu_i[r_i\mu_i] \mid \text{the sub-term } f(\tilde{r}) \text{ unifies with the left-hand side of a rule } f(\tilde{\mu}) \rightarrow r; \text{ with mgu } \mu_i \}.
\]

Here it is supposed that variables are renamed apart during unification. The unifiable rules \( f(\tilde{\mu}) \rightarrow r \) are called narrowing-rules for brevity. Notice that the rule \( l\mu_i \rightarrow r\mu_i[r_i\mu_i] \) corresponds to the specialization of \( l \rightarrow r[f(\tilde{r})] \) by \( \mu_i, \) with \( f \) inlined.
Example 2 (Continued from Example 1). We can narrow rule (b) with narrowing-rules (c) and (d), using the unifiers \{l \mapsto \text{nil}\} and \{l \mapsto \text{cons}(x, x)\}, resulting in the rules
\[
\text{fix[walk]} \cdot \text{nil} \rightarrow \text{lam2} \quad \text{fix[walk]} \cdot \text{cons}(x, x) \rightarrow \text{lam1}_0 \cdot (\text{fix[walk]} \cdot x) \cdot \text{lam3}(x).
\]
Applying rule-narrowing on the second rule twice, we obtain the system consisting of the following six rules.

(1) \quad \text{main}(l) \rightarrow \text{fix[walk]} \cdot l \cdot \text{nil}
(2) \quad \text{fix[walk]} \cdot \text{nil} \rightarrow \text{lam2}
(3) \quad \text{fix[walk]} \cdot \text{cons}(x, x) \rightarrow \text{lam1}_2(\text{fix[walk]} \cdot x, \text{lam3}(x))
(4) \quad \text{lam2} \cdot y \rightarrow y
(5) \quad \text{lam3}(x) \cdot y \rightarrow \text{cons}(x, y)
(6) \quad \text{lam1}_2(f, g) \cdot z \rightarrow f \cdot (g \cdot z).

Notice that the inlined rules (c),(d),(g) and (h), which are no longer used in a reduction of \text{main}, have been dropped.

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 \( l \rightarrow r[f](\bar{t}) \), reachable from a call to \text{main}, is a redex to a generated rule \( l\mu \rightarrow r\mu[\bar{t}] \). 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 \( f(\bar{t}) \) occurring in a reduction in any reduction starting from \text{main}(d_1, \ldots, d_n) (d_j \in \text{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 \( l(\bar{t}_i) \rightarrow r_i \), each variable occurs in the right-hand side \( r_i \) at least as often as it occurs in the left-hand side \( l\sigma(\bar{t}_i) \). We call such rules \text{redex-preserving}. Then the following can be shown.

Proposition 2. Let \( S \) be obtained from the non-overlapping TRS \( \mathcal{R} \) by narrowing the rule \( l \rightarrow r[f](\bar{t}) \) as outlined above. If \( f \) is sufficiently defined and all narrowing-rules are \text{redex-preserving}, then every derivation \text{wrt.} \( \mathcal{R} \) starting from \text{main}(d_1, \ldots, d_n) (d_j \in \text{Data}) of length \( \ell \) is simulated by a derivation \text{wrt.} \( S \) from \text{main}(d_1, \ldots, d_n) of length \( \ell \cdot \lambda \).

Instantiation. Usually the dependency pair transformation, a form of call-graph analysis, is performed as a first step by termination or complexity analysers (see \cite{5,9,2}). This method is ineffective on the applicative systems we generate. To overcome this situation, we intend to transform applicative terms \( f(l_1, \ldots, l_m) \cdot l_{m+1} \cdots l_{m+n} \) to functional form \( f_n(l_1, \ldots, l_m, l_{m+1}, \ldots, l_{m+n}) \). It is well understood how to bring an ATRS into functional form, in a step preserving manner \cite{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 \( S \) a refinement of an ATRS \( \mathcal{R} \) if every rule \( l' \rightarrow r' \in \mathcal{S} \) is an instance of a rule \( l \rightarrow r \in \mathcal{R} \), i.e., \( l' = l\sigma \) and \( r = r'\sigma \) for some substitution \( \sigma \). It is a safe refinement if for every term \( t \) occurring in a derivation \text{wrt.} \( \mathcal{R} \) from \text{main}(d_1, \ldots, d_n) (d_j \in \text{Data}), every redex in \( t \) \text{wrt.} \( \mathcal{R} \) is also a redex \text{wrt.} \( S \). The following proposition then follows by definition.

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

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 \cite{5} 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 \( G \), the set of terms generated from the non-terminal \( X_i \) approximates the assignments to \( x \).

\footnote{Our tool \text{HOCA} reduces the rewrite system automatically to the set of usable rules, an idea that goes back to \cite{1}.}
The regular tree grammar depicted in Figure 1 is obtained from Jones flow analysis. From the productions $F$ rule-narrowing to simplify the right-hand sides:

$$
\begin{align*}
F & \to \text{fix} \cdot \text{walk} \cdot L_6 \cdot \text{nil} | R_2 \cdot \text{nil} | R_4 \cdot \text{nil} | R_6 \\
S_t & \to \text{main}(\ast, \ast) \\
L_1 \cdot X_3 \cdot X_5 \cdot Z_6 & \to \ast \\
Y_S \cdot Z_6 & \to R_4 | \text{nil} \\
Y_S \cdot Z_6 & \to Z_6 \\
F_6 & \to R_2 | R_3 \\
G_6 & \to \text{lam3}(X_3)
\end{align*}
$$

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:

$$
(6a) \quad \text{lam1}_2(\text{lam2}(\text{lam3}(x))) \cdot z \to \text{cons}(x, z) \\
(6b) \quad \text{lam1}_2(\text{lam1}_2(\text{f}, \text{lam3}(y)), \text{lam3}(x)) \cdot z \to \text{lam3}(\text{f}, \text{lam3}(y)) \cdot \text{cons}(x, z).
$$

The regular tree grammar depicted in Figure 7 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 $\text{lam2}$ or $\text{lam1}_2(\text{f}, \text{lam3}(y))$ and $\text{lam3}(x)$, 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 TCT for termination, and TCS 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 applicable term rewrite systems obtained by Proposition 6 in the second column we integrate the techniques of Section 3 and uncurrying 7.

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 similar 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