Exact Flow Analysis by Higher-Order Model Checking

Yoshihiro Tobita, Takeshi Tsukada, and Naoki Kobayashi

Tohoku University, Japan

Abstract. We propose a novel control flow analysis for higher-order functional programs, based on a reduction to higher-order model checking. The distinguished features of our control flow analysis are that, unlike most of the control flow analyses like k-CFA, it is *exact* for simply-typed λ-calculus with recursion and finite base types, and that, unlike Mossin's exact flow analysis, it is indeed runnable in practice, at least for small programs. Furthermore, under certain (arguably strong) assumptions, our control flow analysis runs in time cubic in the size of a program. We formalize the reduction of control flow analysis to higher-order model checking, prove the correctness, and report preliminary experiments.

1 Introduction

Control flow analysis (CFA) is among the most important and popular static analyses for functional programs. It computes a set of functions that may be called from each call site, and the result of CFA is used as a basis of more complex static analyses and also for compiler optimizations. Various CFA methods, with varying degrees of precision and efficiency, have been proposed, including Shiver's k-CFA [22]. Most of the existing CFA algorithms compute an *over-approximation* of the control flow set, even when the target language is restricted to a decidable fragment. To our knowledge, the only exception is Mossin's exact analysis [17], which is however impractical and not runnable in practice.

We propose a novel control flow analysis based on *higher-order model checking* [18, 9]. We reduce CFA to a decision problem on the tree generated by a simply-typed, higher-order functional program, which can be decided by using higher-order model checking. Our method has the following nice properties. First, like Mossin's analysis [17] (and unlike other CFAs), our CFA is *exact* for simply-typed λ-calculus with recursion and *finite* base types.[1] Secondly, unlike Mossin's [17], our CFA is runnable despite its extreme precision, thanks to the recent advances in higher-order model checking [7, 11]. Thus, even if our analysis is still too slow compared with the state-of-the-art CFA methods, it may be useful for evaluating and comparing the precision of other methods. Thirdly, under the (rather strong) assumptions that the largest size of types is fixed and

[1] With infinite base types like integers, our CFA is of course inexact, as the language becomes Turing-complete.

that the nesting depth of function definitions is fixed, our CFA runs in time cubic in the size of a program. Lastly, like Heintze and McAllester's subtransitive CFA [6], our CFA is on-demand, in the sense that it can answer each flow query: "May a function created at ℓ_1 be called at the call site ℓ_2?" in linear time (with the same assumptions as above). Thus, one can invoke our CFA only for critical flow queries that cannot be answered by a faster (but more imprecise) CFA.

Our CFA based on higher-order model checking has been already hinted by Kobayashi [9, 10], but it has not been formalized before. The contributions of the present paper include the formalization, a proof of its correctness, implementation and preliminary experiments.

The rest of this paper is structured as follows. Section 2 reviews higher-order model checking, specialized for the purpose of this paper. Section 3 introduces the source language and defines CFA. Section 4 shows the reduction from CFA to higher-order model checking. Section 5 describes extensions of our CFA. Section 6 reports experiments. Section 7 discusses related work, and Section 8 concludes.

2 Review of Higher-Order Model Checking

This section briefly reviews (a subclass of) higher-order model checking problems (HO model checking, for short) [18, 9], to which we reduce CFA in Section 4. The aim of HO model checking is to check whether the tree generated by a given program satisfies a given property. The usual HO model checking uses *higher-order recursion schemes* [18] as the target, but we consider here a simply-typed λ-calculus with recursion and tree constructors, called λ_T. For tree-generating programs, λ_T has the same expressive power as higher-order recursion schemes.

Let Σ be a finite set of tree constructors. We write $\mathtt{a}$ for an element of Σ, and $\Sigma(\mathtt{a})$ for its arity (which is a non-negative integer). The set of λ_T-*terms* is given by the grammar: $e ::= \mathtt{a} \mid x \mid \mathtt{fun}(f, x, e) \mid e_1 e_2$, where f and x range over variables. The term $\mathtt{fun}(f, x, e)$ represents a recursive function f defined by the equation $f(x) = e$. The term constructor $\mathtt{fun}(f, x, \_)$ binds f and x. As usual, we implicitly rename bound variables as necessary. We write $\lambda x.e$ for $\mathtt{fun}(f, x, e)$ if f does not occur in e.

We assume that the terms are well-typed in the standard simple type system, where there is only a single base type o for trees, and a tree constructor of arity n has type $\underbrace{\mathtt{o} \to \cdots \to \mathtt{o}}_{\Sigma(\mathtt{a})} \to \mathtt{o}$. See [24] for more details. We call a closed term (i.e. a term containing no free variables) of type o a *tree-generating program.*

The operational semantics of λ_T is the standard call-by-name semantics, which evaluates a program to a (possibly infinite) tree: see [24]. Actually we are only concerned with the set of *paths* of the tree generated by a program. Thus, we introduce an alternative semantics for describing the paths. We define $e \xrightarrow{l} e'$ as the least relation satisfying (i) $(\mathtt{fun}(f, x, e))e' \xrightarrow{\epsilon} [e'/x, \mathtt{fun}(f, x, e)/f]e$, (ii) $\mathtt{a}\, e_1 \cdots e_{\Sigma(\mathtt{a})} \xrightarrow{\mathtt{a}} e_i$ for every $i \in \{1, \ldots, \Sigma(\mathtt{a})\}$, and (iii) $e_1 \xrightarrow{l} e_1'$ implies $e_1 e_2 \xrightarrow{l} e_1' e_2$.

The *path language* generated by a program e, written $Path(e)$, is defined by:

$$Path(e) = \{l_1 \cdot \ldots \cdot l_m \mid e \xrightarrow{l_1} e_1 \xrightarrow{l_2} \cdots \xrightarrow{l_m} e_m\}.$$

Here, $\cdot$ denotes the concatenation and ϵ is treated as an empty sequence.

Example 1. Let F be $\texttt{fun}(f, x,\ \texttt{a}\ x\ (f\,(\texttt{b}\ x)))$ and e be $F\ \texttt{c}$. Types for tree constructors and variables are given by $\texttt{a}\colon \texttt{o} \to \texttt{o} \to \texttt{o}$, $\texttt{b}\colon \texttt{o} \to \texttt{o}$, $\texttt{c}\colon \texttt{o}$, $f\colon (\texttt{o} \to \texttt{o})$, and $x\colon \texttt{o}$. Then e has the following reduction sequence:

$$e = F\ \texttt{c} \xrightarrow{\epsilon} \texttt{a}\ \texttt{c}\ (F(\texttt{b}\,\texttt{c})) \xrightarrow{\texttt{a}} F\,(\texttt{b}\,\texttt{c}) \xrightarrow{\epsilon} \texttt{a}\ (\texttt{b}\,\texttt{c})\ (F\,(\texttt{b}\,(\texttt{b}\,\texttt{c}))) \xrightarrow{\texttt{a}} \texttt{b}\,\texttt{c} \xrightarrow{\texttt{b}} \texttt{c}\ .$$

Thus $\varepsilon, \texttt{a}, \texttt{aa}, \texttt{aab} \in Path(e)$. The path language $Path(e)$ is $\{\texttt{a}^n\texttt{b}^m \mid n > m \geq 0\} \cup \{\varepsilon\}$, where $\texttt{a}^n$ is a sequence of length n consisting of $\texttt{a}$. □

We are interested in the decision problem: "given a program e and a regular language R, is $Path(e)$ a subset of R?" It can be considered an instance of higher-order model checking, and its decidability follows from Ong's result [18].

Theorem 1. *Let e be a tree-generating program and R a regular word language. Then whether $Path(e) \subseteq R$ is decidable.*

In the rest of this paper, we just call the decision problem $Path(e) \overset{?}{\subseteq} R$ above a *HO model checking problem.* It can be solved by using existing higher-order model checkers [7, 11, 15].[2] Note that higher-order model checking [18, 9] is a generalization of conventional (finite-state or pushdown) model checking, and that finite state model checkers cann The worst-case complexity is in general non-elementary [18, 12]. Under certain assumptions, however, it is linear time in the program size [9]. It is rephrased for our language as follows.

Theorem 2. *Suppose (i) R is fixed and (ii) the largest type size of a variable in e and the nesting depth of function definitions are bounded above by a constant. Then, $Path(e) \overset{?}{\subseteq} R$ can be decided in time linear in the size of e.*

The constant factor is, however, huge: It is non-elementary in the parameters that have been assumed to be bounded by constants above.

3 Source Language and CFA Problem

3.1 Source Language λ_S

Our source language λ_S is a simply-typed λ-calculus extended with recursions and non-deterministic branches. Its syntax is defined by:

$$t \text{ (terms)} ::= v \mid x \mid t_1 \mathbin{@^\ell} t_2 \mid \texttt{if*}\ t_1\ t_2 \qquad v \text{ (value)} ::= \texttt{()} \mid \texttt{fun}^\ell(f, x, t)$$
$$T \text{ (types)} ::= \texttt{Unit} \mid T_1 \ \to\ T_2.$$

[2] Higher-order model checking [18, 9] should not be confused with ordinary (finite state) model checking. The former can be considered a generalization of the latter.

Here, $\mathtt{fun}^{\ell}(f, x, t)$ describes a recursion function f given by $f(x) = t$. We often write $\lambda^{\ell}x.t$ for $\mathtt{fun}^{\ell}(f, x, t)$ when f does not occur in t. The term $t_1 \;@^{\ell}\; t_2$ applies the function t_1 to t_2, and $\mathtt{if*}\; t_1\; t_2$ reduces to t_1 or t_2 in a non-deterministic manner. The non-determinism will be used to abstract values in Section 5. To talk about flows of functions, we attach a label ℓ to each function and application. We use a special dummy label $\ell_\star$ for an unimportant label and often omit it. We write $\mathcal{L}$ for the set of all labels, and $\mathcal{L}^-$ for $\mathcal{L} \setminus \{\ell_\star\}$.

The evaluation strategy of λ_S is call-by-value[3]. The evaluation context (E) and the reduction relation ($\longrightarrow$) are defined by:

$$E \text{ (evaluation contexts) } ::= [\,] \mid E@^{\ell}t \mid v@^{\ell}E.$$

$$\begin{array}{c}
E[\mathtt{fun}^{\ell}(f, x, t_1)@^{\ell'}v_2] \;\longrightarrow\; E[[v_2/x,\ \mathtt{fun}^{\ell}(f, x, t_1)/f]t_1] \\
E[\mathtt{if*}\; t_1\; t_2] \;\longrightarrow\; E[t_1] \qquad E[\mathtt{if*}\; t_1\; t_2] \;\longrightarrow\; E[t_2].
\end{array}$$

Here, $E[t]$ is the expression obtained by replacing the hole $[\,]$ in E with t, and $[t_1/x_1, \ldots, t_k/x_k]t$ denotes the term obtained by replacing every free occurrence of x_i in t with t_i. We write $\longrightarrow^+$ for the transitive closure and $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

As usual, the type judgment relation $\Gamma \vdash t : T$ is defined as the least relation closed under the rules below. We call a closed λ_S-term of type $\mathtt{Unit}$ (i.e., a term t such that $\emptyset \vdash t : \mathtt{Unit}$) a *source program.*

$$\frac{}{\Gamma \vdash \mathtt{()} : \mathtt{Unit}} \qquad \frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash \mathtt{if*}\; t_1\; t_2 : T_1}$$

$$\frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash t_1 : T_2}{\Gamma \vdash \mathtt{fun}^{\ell}(f, x, t_1) : T_1 \to T_2} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \;@^{\ell}\; t_2 : T_2}$$

3.2 CFA Problem

We define CFA as a decision problem to check whether a given function may be called at a given call site.

Definition 1 (Control-flow relation and CFA problem). *For a source program t, the* control-flow relation *$CF(t)$ is given by:*

$$CF(t) = \{(\ell_1, \ell_2) \in \mathcal{L}^- \times \mathcal{L}^- \mid \exists t_1, v_2, f, x, E.\ t \longrightarrow^* E[(\mathtt{fun}^{\ell_2}(f, x, t_1))@^{\ell_1}v_2]\}.$$

CFA *is the problem of deciding whether $(\ell_1, \ell_2) \in CF(t)$, for a given program t and labels $\ell_1, \ell_2 \in \mathcal{L}^-$.*

A usual flow analysis aims to compute an *over-approximation* of the set $CF(t)$. In the next section, we shall give an *exact* flow analysis algorithm for solving the above decision problem. It is non-trivial even for λ_S, as it has recursion and non-determinism.

[3] We can also deal with CFA for call-by-name languages, just by changing the CPS transformation in Section 4.1

Example 2. Consider the program $t_p = (\lambda^1 x.\ x@^2\texttt{()})@^3(\lambda^4 z.\texttt{()})$. It is evaluated as: $(\lambda^1 x.\ x@^2\texttt{()})@^3(\lambda^4 z.\texttt{()}) \longrightarrow (\lambda^4 z.\texttt{()})@^2\texttt{()} \longrightarrow \texttt{()}$. The function (labeled by) 1 is applied at the call site 3, and the function 4 is applied at the call site 2. By the definition, $CF(t_p) = \{(3,1),(2,4)\}$.

Remark 1. The CFA problem above is defined only for closed programs. CFA is undecidable for open programs containing higher-order variables [19].

4 Reduction from CFA to HO Model Checking

This section reduces CFA to the HO model checking problem reviewed in Section 2. By combining the reduction with a HO model checking algorithm we obtain a sound and complete algorithm for CFA. The reduction consists of two steps: we first reduce CFA to *call-sequence analysis* (CSA), which is the problem of analyzing the order of function call. We then reduce CSA to HO model checking, i.e., a verification problem for a tree-generating program.

4.1 From CFA to CSA

We first define the CSA problem.

Definition 2 (CSA problem). *For a term t, we define $CS(t)$ by:*

$$\begin{aligned} CS(t) = \{&(\ell_1,\ell_2) \in \mathcal{L}^- \times \mathcal{L}^- \mid \\ &t \longrightarrow^* E_1[(\textit{\textbf{fun}}^{\ell_1}(f_1,x_1,t_1)@v_1] \longrightarrow E_2[(\textit{\textbf{fun}}^{\ell_2}(f_2,x_2,t_2)@v_2]\}. \end{aligned}$$

CSA *is the problem of deciding whether $(\ell_1,\ell_2) \in CS(t)$, for a given term t and labels $\ell_1,\ell_2 \in \mathcal{L}^-$.*

Example 3. Let us consider the following program t_0.

$$t_0 = (\lambda^3 b.(\lambda^1 x.\lambda k_1.(\lambda^2 a.\ x\ a\ k_1)\ \texttt{()})\ b\ (\lambda m.m))\ (\lambda^4 z.\lambda k_2.\ k_2\ \texttt{()}).$$

We have omitted symbols @ and their labels for readability. t_0 is reduced as:

$$\begin{aligned} &(\lambda^3 b.(\lambda^1 x.\lambda k_1.(\lambda^2 a.x\ a\ k_1)\ \texttt{()})\ b\ (\lambda m.m))\ (\lambda^4 z.\lambda k_2.\ k_2\ \texttt{()}) \\ &\longrightarrow (\lambda^1 x.\lambda k_1.(\lambda^2 a.\ x\ a\ k_1)\ \texttt{()})\ (\lambda^4 z.\lambda k_2.\ k_2\ \texttt{()})\ (\lambda m.m) \\ &\longrightarrow^* (\lambda^2 a.(\lambda^4 z.\lambda k_2.\ k_2\ \texttt{()})\ a\ (\lambda m.m))\ \texttt{()} \\ &\longrightarrow (\lambda^4 z.\lambda k_2.\ k_2\ \texttt{()})\ \texttt{()}\ (\lambda m.m) \longrightarrow^* \texttt{()}. \end{aligned}$$

Thus, $CS(t_0) = \{(3,1),(2,4)\}$. □

To reduce CFA to CSA, it suffices to apply the following call-by-value continuation-passing-style (CPS) transformation $[\![\cdot]\!]$ [2, 20].

$$\begin{aligned} [\![\texttt{()}]\!] &= \lambda^{\ell_\star} k.k\ @^{\ell_\star}\texttt{()} \\ [\![x]\!] &= \lambda^{\ell_\star} k.k\ @^{\ell_\star} x \\ [\![\texttt{fun}^\ell(f,x,t)]\!] &= \lambda^{\ell_\star} k.k\ @^{\ell_\star}(\texttt{fun}^\ell(f,x,[\![t]\!])) \\ [\![t_1\ @^\ell t_2]\!] &= \lambda^{\ell_\star} k.[\![t_1]\!]\ @^{\ell_\star}(\lambda^{\ell_\star} f.[\![t_2]\!]@^{\ell_\star}(\lambda^\ell z.\ (f\ @^{\ell_\star} z)\ @^{\ell_\star} k))\ (f,z \text{ are fresh}) \\ [\![\texttt{if*}\ t_1\ t_2]\!] &= \lambda^{\ell_\star} k.(\texttt{if*}\ ([\![t_1]\!]@^{\ell_\star} k)\ \ ([\![t_2]\!]@^{\ell_\star} k)). \end{aligned}$$

$$
\begin{array}{ll|ll}
 & t & & \llbracket t \rrbracket @ (\lambda m.m) \\
\longrightarrow^* & E[t_1 \ @^{\ell_1} t_2] & \longrightarrow^* & \llbracket t_1 @^{\ell_1} t_2 \rrbracket @ K \\
\longrightarrow^* & E[(\lambda^{\ell_2} x.\ t_3) \ @^{\ell_1} t_2] \quad \text{(i)} & \longrightarrow^* & \llbracket t_2 \rrbracket \ @ \ (\lambda^{\ell_1} z.((\lambda^{\ell_2} x.\llbracket t_3 \rrbracket)@z)@K) \ \ \text{(i)} \\
\longrightarrow^* & E[(\lambda^{\ell_2} x.\ t_3) \ @^{\ell_1} v_4] \ \ \text{(ii)} & \longrightarrow^* & (\lambda^{\ell_1} z.((\lambda^{\ell_2} x.\llbracket t_3 \rrbracket)@z@K)) \ @ \ \llbracket v_4' \rrbracket \ \text{(ii)} \\
 & & \longrightarrow & (\lambda^{\ell_2} x.\llbracket t_3 \rrbracket)@\llbracket v_4' \rrbracket @K
\end{array}
$$

Fig. 1. Evaluation of an application on source and CPS programs

By the CPS transformation, every term is converted to a function that takes a continuation (i.e., the rest of the computation) and passes the evaluation result to it. The CPS transformation above is the same as the standard (simplest) one [20], except for the treatment of labels. In the third rule, the label ℓ of the function $\texttt{fun}(f, x, t)$ is retained in the result $\texttt{fun}^{\ell}(f, x, \llbracket t \rrbracket)$. In the fourth rule, the label ℓ of the function application is moved to the continuation argument for $\llbracket t_2 \rrbracket$. Dummy labels are attached to all the other functions introduced by the transformation. Note that the CPS transformation above is type-preserving [16]. If t is a source program (of type $\texttt{Unit}$), then $\llbracket t \rrbracket$ has type $(\texttt{Unit} \rightarrow \texttt{Unit}) \rightarrow \texttt{Unit}$.

The transformation rule for $t_1 @^{\ell} t_2$ above is the key for the reduction from CFA to CSA. In $\llbracket t_1 @^{\ell} t_2 \rrbracket$, ℓ is attached to $\lambda z.f@z@k$, which is the continuation to call the function f obtained by evaluating t_1. Thus, if the value of t_1 is labeled by ℓ_1 in a source program, then ℓ_1 is called immediately after the continuation function $\lambda^{\ell} z.f@z@k$ in the target program. Figure 1 shows a rough correspondence between reduction sequences of a source program and its target program.[4] The left-hand side shows a reduction sequence of a source program t that leads to a call of function ℓ_2 from a call site ℓ_1. t is first evaluated to $E[t_1 @^{\ell_1} t_2]$, and then t_1 is evaluated to a function $\lambda^{\ell_1} x.t_3$ (see the term marked by (i)). t_2 is then evaluated to a value v_4, and the function ℓ_2 is called at ℓ_1 (see (ii)). The corresponding reduction sequence after the CPS transformation is shown on the right-hand side. The term (i) shows the state after t_1 has been evaluated and before t_2 is evaluated. The term (ii) shows the state after t_2 has been evaluated and its value being passed to the continuation. After that, ℓ_1 and ℓ_2 are consecutively called.

From the correspondence above, the following theorem should be intuitively clear. A proof is given in [24].

Theorem 3 (Correctness of the Reduction from CFA to CSA). *Let t be a source program and ℓ_1, ℓ_2 labels in t. Then, $(\ell_1, \ell_2) \in CF(t)$ if and only if $(\ell_1, \ell_2) \in CS(\llbracket t \rrbracket @ (\lambda m.m))$.*

[4] For the sake of simplicity, we show here an inaccurate reduction sequence on the right-hand side. See [24] for the exact correspondence.

Example 4. Recall the program t_p in Example 2. After the CPS transformation, the program is reduced as follows (where $K = \lambda m.m$):

$$[\![(\lambda^1 x.\ x@^2\texttt{()})@^3(\lambda^4 y.\texttt{()})]\!]@K \longrightarrow^* (\lambda^3 z.((\lambda^1 x.[\![x@^2\texttt{()}]\!])@z)@K)@(\lambda^4 y.[\![\texttt{()}]\!])$$
$$\longrightarrow ((\lambda^1 x.[\![x@^2\texttt{()}]\!])@(\lambda^4 y.[\![\texttt{()}]\!]))@K \longrightarrow^* (\lambda^2 z'.((\lambda^4 y.[\![\texttt{()}]\!])@z')@K)@\texttt{()}$$
$$\longrightarrow ((\lambda^4 y.[\![\texttt{()}]\!])@\texttt{()})@K \longrightarrow^* \texttt{()}.$$

Thus, $CS([\![t_p]\!]@K) = \{(3,1),(2,4)\}(= CF(t_p))$.

4.2 From CSA to HO Model Checking

To reduce CSA to HO model checking, we transform the output of the CPS transformation into a tree-generating program having tree constructors: $\Sigma = \{\ell \mapsto 1 \mid \ell \in \mathcal{L}\} \cup \{\texttt{br} \mapsto 2, \texttt{e} \mapsto 0\}$. Here, ℓ is a tree constructor for a label, and br is a tree constructor for conditionals. The translation $\langle\!\langle\cdot\rangle\!\rangle$ to the tree-generating program is given by:

$$\langle\!\langle\texttt{()}\rangle\!\rangle = \texttt{e} \quad \langle\!\langle x\rangle\!\rangle = x \quad \langle\!\langle t_1@^\ell t_2\rangle\!\rangle = \langle\!\langle t_1\rangle\!\rangle\ @\ \langle\!\langle t_2\rangle\!\rangle \quad \langle\!\langle \texttt{if}*\ t_1\ t_2\rangle\!\rangle = \texttt{br}\ \langle\!\langle t_1\rangle\!\rangle\ \langle\!\langle t_2\rangle\!\rangle.$$
$$\langle\!\langle \texttt{fun}^\ell(f,x,\lambda k.t)\rangle\!\rangle = \texttt{fun}(f,x,\lambda k.\ell@\langle\!\langle t\rangle\!\rangle) \quad (\text{if } t \text{ has type } \texttt{Unit})$$
$$\langle\!\langle \lambda^\ell x.t\rangle\!\rangle = \lambda x.\ell@\langle\!\langle t\rangle\!\rangle \quad (\text{if } t \text{ has type } \texttt{Unit})$$

Note that the translation above is well-defined for the image of the CPS transformation: A function occurs only in the form: (i) $\texttt{fun}^\ell(f,x,\lambda k.t)$ (which comes from a function in a source program) where t has type Unit, or (ii) $\lambda^\ell x.t$ (which is a continuation function) where t has type Unit. The translation is also type-preserving, turning the base type Unit to o.

Whenever a function $\texttt{fun}^\ell(f,x,\lambda k.t)$ or $\lambda^\ell x.t$ is called in a program in CPS, a tree node labeled with ℓ is created in the corresponding λ_T-program obtained by the above translation. From this observation, it should be clear that CSA has now been reduced to HO model checking, as stated below without a proof.

Theorem 4 (Correctness of the Reduction from CSA to HOMC). *Let t be a λ_S-program, t_C be $[\![t]\!]@(\lambda z.z)$, and ℓ_1, ℓ_2 be labels in t. Then, $(\ell_1,\ell_2) \in CS([\![t]\!]@(\lambda z.z))$ if and only if $\exists w_1, w_2.\ w_1\ell_1\ell_2 w_2 \in Path(\langle\!\langle t_C\rangle\!\rangle)$.*

From Theorems 3 and 4, we obtain the following corollary.

Corollary 1 (Correctness of the Reduction from CFA to HOMC). *Let t be a source program and ℓ_1, ℓ_2 labels in t. Then, $(\ell_1,\ell_2) \in CF(t)$ if and only if $\exists w_1, w_2.\ w_1\ell_1\ell_2 w_2 \in Path(\langle\!\langle [\![t]\!]@(\lambda m.m)\rangle\!\rangle)$.*

4.3 Complexity

We now discuss the time complexity of our flow analysis with respect the program size. We assume below that both (i) the largest size of the type of a variable and (ii) the nesting depth of function definitions are bounded above by a constant.

These are rather strong assumptions, but in realistic programs, these parameters do not seem to depend much on the program size, so that it would be reasonable to assume that they are constants when we discuss the parameterized complexity with respect to the program size. Heintze and McAllester [6] also assume that the largest type size is bounded by a constant. Without the assumptions above, the time complexity of our CFA is non-elementary in the program size. We write $|t|$ for the size of a program t.

Theorem 5. *Given a program t and labels ℓ_1, ℓ_2, the query $(\ell_1, \ell_2) \overset{?}{\in} CF(t)$ can be answered in time $O(|t|)$, under the assumption above. Under the same assumption, the control flow set $CF(t)$ can be computed in time $O(|t|^3)$.*

Proof. Under the assumption, the reductions from CFA to CSA and from CSA to HO model checking can be carried out in time linear in the size of t, and the size of the tree-generating program is $O(|t|)$. The resulting HO model checking problem satisfies the assumption of Theorem 2, hence solved in linear time. The flow set $CF(t)$ can be computed by deciding $(\ell_1, \ell_2) \overset{?}{\in} CF(t)$ for every pair (ℓ_1, ℓ_2) of labels. As the number of pairs is $O(|t|^2)$, the total cost is $O(|t|^3)$. $\square$

The control flow set $CF(t)$ is often sparse, and its size is much smaller than $|t|^2$. In such a case, we can use a binary search to compute $CF(t)$ in time $O(m|t| \log |t|)$, where m is the size of $CF(t)$. For that purpose, we consider the following extended CFA problem:

"Given $L_1, L_2 (\subseteq \mathcal{L}^-)$ and a program t, is $(L_1 \times L_2) \cap CF(t)$ empty?"

The algorithm for the extended CFA can be obtained by slightly modifying our algorithm for CFA: in the last step from CSA to HO model checking, we just need to replace all the labels in L_1 with the same tree constructor ℓ_1 and those in L_2 with ℓ_2. Under the same assumption as for CFA, the extended CFA query can be answered in time $O(|t|)$.

Figure 2 shows an algorithm to output all elements of $CF(t)$ by using the extended CFA. In the figure, $L_@$ (L_λ, resp.) is the set of all labels attached to call sites (functions, resp.) in t. The function `div` splits a set L into two disjoint sets L_1 and L_2 such that $L = L_1 \cup L_2$ and $|L_2| \leq |L_1| \leq |L_2| + 1$. `subCF`$(L_1, L_2)$ outputs all the elements of $(L_1 \times L_2) \cap CF(t)$. It first checks whether $(L_1 \times L_2) \cap CF(t)$ is empty. If $(L_1 \times L_2) \cap CF(t)$ is non-empty and L_1 and L_2 are singleton sets, then `subCF`(L_1, L_2) just outputs the flow pair. Otherwise, it divides L_1 or L_2 and calls `subCF` recursively. For each $(\ell_1, \ell_2) \in CF(t)$, the algorithm checks the emptiness of $(L_1 \times L_2) \cap CF(t)$ for some L_1, L_2 such that $(\ell_1, \ell_2) \in L_1 \times L_2$ $O(\log |t|)$ times. Thus, the whole algorithm runs in time $O(m|t| \log |t|)$, provided $m > 0$.

5 Extensions

We have so far considered a simple language having only functions and the unit value. In this section, we discuss how to extend our CFA to deal with other data and control structures. We also discuss an extension to compute *data* flow.

```
      enumCF () = subCF($L_@, L_\lambda$)
subCF($L_1, L_2$) = if $L_1 \times L_2 \cap CF(t) = \emptyset$ then ()
                    else if $(L_1 = \{\ell_1\}) \wedge (L_2 = \{\ell_2\})$ then output $(\ell_1, \ell_2)$
                    else if $|L_1| \leq |L_2|$ then
                        let $(L_{21}, L_{22})$ = div($L_2$) in subCF($L_1, L_{21}$); subCF($L_1, L_{22}$)
                    else let $(L_{11}, L_{12})$ = div($L_1$) in subCF($L_{11}, L_2$); subCF($L_{12}, L_2$)
```

Fig. 2. Algorithm `enumCF` for CFA with the binary search technique.

5.1 Booleans and Control Structures

We can extend our exact CFA to deal with booleans and control structures such as (a restricted form of) exceptions and call/cc, without losing the exactness.

To deal with booleans, we just need to extend the reduction from CFA to CSA, by combining the CPS transformation with Church encoding to enumerate booleans. The extended transformation is given as follows.

$$[\![\texttt{true}]\!] = \lambda k.k@(\lambda t.\lambda f.t) \qquad [\![\texttt{false}]\!] = \lambda k.k@(\lambda t.\lambda f.f)$$
$$[\![\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3]\!] = \lambda k.([\![t_1]\!]@(\lambda b.b@([\![t_2]\!]@k)@([\![t_3]\!]@k))).$$

In the above transformation, we apply the standard CPS transformation (e.g. to transform `true` to $\lambda k.k\texttt{true}$) and then encode booleans into functions (e.g. `true` to $\lambda t.\lambda f.t$). The result of the transformation is a well-typed λ_S-program: an expression of type `Bool` is transformed to that of type $((\texttt{o} \to \texttt{o} \to \texttt{o}) \to \texttt{o}) \to \texttt{o}$.

As our analysis is precise for higher-order functions, we can also deal with control structures such as (i) a finite number of exceptions that do not carry values and (ii) (the simply-typed version of) call/cc of type $((\tau \to \texttt{Unit}) \to \tau) \to \tau$ by encoding them in λ_S. Exceptions can be encoding by using auxiliary continuations [1]. For `call/cc`, it suffices to extend the transformation by:

$$[\![\texttt{call/cc } t]\!] = \lambda k.[\![t]\!]@(\lambda f.f@(\lambda x.\lambda k'.k@x)@k).$$

5.2 Infinite Data Domains

If λ_S is extended with infinite data domains such as integers and lists, the CFA problem becomes undecidable. Thus, we have to give up the exact analysis and apply some abstraction to compute an over-approximation of the actual flow set. The simplest solution is to ignore all the values except functions and booleans and replace them with unit values, before applying our CFA. Such a translation $[\![\cdot]\!]_S$ from the extended language to λ_S is given by:

$$[\![\texttt{Int}]\!]_S = \texttt{Unit} \quad [\![\tau_1 \to \tau_2]\!]_S = [\![\tau_1]\!]_S \to [\![\tau_2]\!]_S \quad [\![\tau\ \texttt{List}]\!]_S = \texttt{Unit} \to [\![\tau]\!]_S$$
$$[\![n]\!]_S = \texttt{()} \text{ if } n \text{ is an integer}$$
$$[\![+]\!]_S = \lambda x.\lambda y.\texttt{()} \quad [\![=_{\texttt{Int}}]\!]_S = \lambda x.\lambda y.\texttt{if}{*}\ \texttt{true false}$$
$$[\![\texttt{nil}]\!]_S = \texttt{fun}(f, x, f@x) \quad [\![\texttt{cons}]\!]_S = \lambda x.\lambda l.\lambda z.\texttt{if}{*}\ x\ (l@\texttt{()})$$
$$[\![\texttt{hd}]\!]_S = \lambda l.l@\texttt{()} \qquad [\![\texttt{tl}]\!]_S = \lambda l.l$$
$$[\![x]\!]_S = x \quad [\![t_1@^\ell t_2]\!]_S = [\![t_1]\!]_S@^\ell[\![t_2]\!]_S \quad [\![\texttt{fun}^\ell(f, x, t)]\!]_S = \texttt{fun}^\ell(f, x, [\![t]\!]_S)$$

Here, a list of elements of type τ is represented by a non-deterministic function that takes the unit value as an argument and returns an element of the list.

Example 5. Let t_l be $(\texttt{hd}\ (\texttt{cons}\ (\lambda^1 x.\texttt{()})\ (\texttt{cons}\ (\lambda^2 y.y)\ \texttt{nil})))\ @^3\texttt{()}$. $[\![t_l]\!]_S$ is reduced to $(\texttt{if}*\ (\lambda^1 x.\texttt{()})\ ((\lambda u.\texttt{if}*\ (\lambda^2 y.y)\ (\texttt{fun}(f, x,\ f@\texttt{()})))\ @\ \texttt{()}))\ @^3\texttt{()}$, which is then non-deterministically reduced to one of the following terms:

$$(\lambda^1 x.\texttt{()})\ @^3\texttt{()} \qquad (\lambda^2 y.y)\ @^3\texttt{()} \qquad (\texttt{fun}(f, x,\ f@\texttt{()}))\ @^3\texttt{()}$$

Thus, $\mathit{CF}([\![t_l]\!]_S) = \{(3,1),(3,2)\}$, which is an over-approximation of the actual flow set $\mathit{CF}(t_l) = \{(3,1)\}$. □

Though information about the order of list elements is lost, our analysis is still very precise compared with 0CFA, as demonstrated in the following example.

Example 6. Consider the following program:

$$\texttt{let}\ \mathit{app} = \lambda(f, x).f@x\ \texttt{in map}\ \mathit{app}\ [(f_1, g_1); \cdots; (f_n, g_n)]$$

where $f_i = \lambda h.h@^{\ell_i}\texttt{()}$. Our method (extended to handle pairs) can infer that only g_i (not g_j for $j \neq i$) may be called at ℓ_i. □

The abstraction above completely throws away information about values other than functions and booleans. A more precise analysis can be obtained by using predicate abstractions [13].

5.3 Data Flow Analysis

We can extend our CFA to *data* flow analysis (DFA), which computes (an over-approximation of) the flow of not only functions but other data. We add a label to each sub-expression of the program. The DFA problem is then defined as a decision problem: "Given a program t and labels ℓ_1, ℓ_2, may an expression labeled by ℓ_2 evaluate to a value created at program point ℓ_1?". We write $\mathit{DF}(t)$ for the set of all pairs (ℓ_1, ℓ_2) that satisfies the condition. The DFA problem is undecidable in general in the presence of infinite data domains. Thus the goal here is to compute an over-approximation of the set $\mathit{DF}(t)$.

Example 7. Consider the program: $(\lambda x.\lambda y.x^{\ell_{d,0}} + y^{\ell_{d,1}})@1^{\ell_{s,0}}@2^{\ell_{s,1}}$. Here, we have given a label $\ell_{s,i}$ for a source of data flow, and $\ell_{d,j}$ for a destination. $\mathit{DF}(t) = \{(\ell_{s,0}, \ell_{d,0}), (\ell_{s,1}, \ell_{d,1})\}$. □

(An over-approximation of) $\mathit{DF}(t)$ can be computed by a reduction to CFA, encoding all data into functions. For integers, we modify the encoding $[\![\cdot]\!]_S$ in Section 5.2 as follows.

$$[\![\texttt{Int}]\!]_{S'} = \texttt{Unit} \to \texttt{Unit}$$
$$[\![n^{\ell_s}]\!]_{S'} = \lambda x^{\ell_s}.\texttt{()} \text{ if } n \text{ is an integer} \qquad [\![+^{\ell_s}]\!]_{S'} = \lambda x.\lambda y.\lambda z^{\ell_s}.\texttt{()}$$
$$[\![e^{\ell_d}]\!]_{S'} = \texttt{let}\ x = [\![e]\!]_{S'}\ \texttt{in}\ (x@^{\ell_d}\texttt{()}; x)\ (e \text{ has type } \texttt{Int})$$

Here, both $\texttt{let } x = e_1 \texttt{ in } e_2$ and $e_1; e_2$ are abbreviations for $(\lambda x.e_2)@e_1$ where the latter is a special case of the former, when x does not occur in e_2. The label ℓ_s attached to $+$ expresses the value created by the operation. An integer is turned into a function labelled by its creation point. For an expression with a destination label ℓ_d, we insert an application labeled with ℓ_d. Then, it should be obvious that if $(\ell_s, \ell_d) \in DF(t)$ then $(\ell_d, \ell_s) \in CF([\![t]\!]_{S'})$.

Example 8. Recall the program in Example 7. It is translated to the following λ_S-program $[\![t]\!]_{S'}$:

$$(\lambda x.\lambda y.(\underbrace{\lambda x'.\lambda y'.\lambda z^{\ell_{s,2}}.\texttt{()}}_{+})\,\underbrace{(x@^{\ell_{d,0}}\texttt{()}; x)}_{x^{\ell_{d,0}}}\,\underbrace{(y@^{\ell_{d,1}}\texttt{()}; y)}_{y^{\ell_{d,1}}})@\underbrace{(\lambda x^{\ell_{s,0}}.\texttt{()})}_{1^{\ell_{s,0}}}@\underbrace{(\lambda x^{\ell_{s,1}}.\texttt{()})}_{2^{\ell_{s,1}}}$$

Here we have inlined let-expressions in $[\![x^{\ell_{d,0}}]\!]_{S'}$ and $[\![y^{\ell_{d,1}}]\!]_{S'}$. By using our CFA, we obtain $CF([\![t]\!]_{S'}) = \{(\ell_{d,0}, \ell_{s,0}), (\ell_{d,1}, \ell_{s,1})\}$. Thus, we know $DF(t) \subseteq \{(\ell_{s,0}, \ell_{d,0}), (\ell_{s,1}, \ell_{d,1})\}$.

6 Experiments

We have implemented our flow analysis for the extension of λ_S with integers and lists, as discussed in Section 5. The current implementation analyzes the flow of functions and integers. For data flow analysis, argument positions of integer operations $(+, =, <, \ldots)$ are taken as destinations of data flow. TRecS [7, 8] is used as the underlying model checker.[5]

The results of preliminary experiments are summarized in Table 1. In the table, the column OS shows the largest order of types in a source program, where the order of a type is defined by: $\mathit{order}(\texttt{Unit}) = \mathit{order}(\texttt{Int}) = 0$ and $\mathit{order}(T_1 \to T_2) = \mathit{max}(\mathit{order}(T_1) + 1, \mathit{order}(T_2))$. OT shows the largest order of types in the tree-generating program (represented in the form of higher-order recursion schemes [18]) obtained by the two step reductions. For comparison, the column "0CFA" shows the number of flow queries for which 0CFA answered yes. As 0CFA outputs an over-approximation of the actual flow set, it is always greater than or equal to the number in the column "Flow". The numbers in parentheses show the number of queries among them for which TRecS timed out. The first five programs `fib`–`tak` have been taken (and slightly modified) from the benchmark set of MLton (http://mlton.org/Performance), and `callcc` has been taken from [26], obtained by encoding call/cc. The other programs have been handcrafted by ourselves. For space restriction, we explain only some of the programs below: see [24] for more details. `app` defines an apply function, and uses it twice for different pairs of functions and arguments:

```
let apply = fun f -> fun g -> f g in
let f1 = .. in let f2 = .. in let g1 = .. in let g2 = .. in
(apply f1 g1)+(apply f2 g2)
```

[5] There are a few other higher-order model checkers [11, 15] to date, but TRecS appears to be the fastest for this type of application.

Our analysis is able to infer that `f1` is applied to `g1` and `f2` is applied to `g2`, unlike 0CFA. `app_div` is the same as above, except that the definition of `apply` has been replaced by:

```
let rec apply' = fun f -> fun g -> (f g; apply' f g) in ...
```

Our analysis respects the call-by-value semantics and correctly infers that `f2` is never called. `map` creates a list of integers and applies a function to each element. `map_pair` is the program discussed in Example 6. `map_pair2` is the same as `map_pair`, except that the definition of `map` is optimized to:

```
let map f l = if null(l) then nil else f(car(l)),
```

by taking into account our encoding of list primitives: note that, due to the over-approximation introduced by our encoding, the flow set remains the same.

Some observations from the results in Table 1 follow. First, as expected, the analysis is very slow, compared with the state-of-the-art control flow analyzer (e.g., see [21]). In fact, our naive implementation of 0CFA terminated in less than 0.1 second for all the benchmark programs. This point may however be improved by refining higher-order model checkers and the reduction from CFA to higher-order model checking. Secondly, for many of the tested programs, all the flow queries were answered by TRecS, which is encouraging given the extremely high worst-case complexity (i.e., k-EXPTIME-completeness for order-k programs) of higher-order model checking [18]. TRecS [7] does not always suffer from the k-EXPTIME bottleneck, and tends to terminate quickly if there is a small certificate for a verified property. The result suggests that for our benchmark programs, certificates of flow or non-flow are small enough. Three flow queries timed out: one for `merge` and two for `tak`. These are due to a limitation of the current TRecS, rather than that of our approach. In fact, for `tak`, those queries are answered "yes" if a parameter of the model checker TRecS is manually adjusted. The query for `merge` is answered "no" by another higher-order model checker under development. For `imp_for`, `app_div`, and `map_pair2`, we can confirm that our analysis is more accurate than 0CFA. Experimental comparison with more precise analyses such as k-CFA and CFA2 is left for future work.

7 Related Work

A number of methods for control flow analysis have been studied, including k-CFA [22], polymorphic splitting [27], type-based flow analyses [6, 17, 4], and CFA2 [26, 25]. To our knowledge, ours is the first implementation of a flow analysis that is *exact* for the simply-typed λ-calculus with recursion. Except Mossin's exact flow analysis [17], the previous methods are not exact even for λ_S in Section 3. An advantage brought by the exactness of our analysis for higher-order functions is that various control structures can be easily handled via encoding without losing any precision, as discussed in Section 5.1. This is in contrast

Program	OS	OT	Call	Fun	Dest	Op	Const	Time	Flow	No Flow	TimeOut	0CFA
fib	1	4	7	3	20	8	7	5.56	42	279	0	42
merge	1	8	20	10	29	13	6	85.38	60	690	1	62 (1)
mandelbrot	1	8	18	12	49	22	8	276.94	83	1603	0	83
tailfib	1	8	13	8	17	7	7	3.33	36	306	0	36
tak	1	8	19	6	14	6	6	125.34	58	222	2	60 (2)
callcc	4	5	3	2	7	3	2	0.16	0	41	0	—
imp_for	2	9	33	17	8	3	4	13.63	47	570	0	61
app_div	3	8	9	6	4	2	2	0.14	6	64	0	15
map	2	8	6	3	8	4	4	1.25	21	61	0	21
map_imflist	2	8	9	6	3	2	4	1.30	13	59	0	13
map_rand	2	10	13	5	15	7	3	15.91	36	179	0	36
map_pair	5	13	15	12	3	2	2	5.80	19	173	0	22
map_pair2	5	13	13	12	3	2	2	0.88	17	151	0	20

Machine spec.: Intel(R) Core(TM)2 Duo 3.16GHz CPU and 3.21GB memory. Columns OS and OT: the order of the source and target programs. Call and Fun: the number of call sites and functions. Dest: the number of destinations of data flow. Op and Const: the number of (occurrences of) integer operations and constants. Time: the total running time (second) of the flow analysis. Flow: the number of flow queries answered "yes". No Flow: the number of flow queries answered "no". TimeOut: the number of flow queries for which TRecS could not answer in 10 seconds. 0CFA: the number of flow queries for which 0CFA answers "yes".

Table 1. Results of experiments.

with CFA2, which needed to be adapted to deal with call/cc [26]. Our reduction from CFA to CSA uses CPS transformation. Incidentally, usefulness of CPS transformation in flow analysis has been already pointed out by Shivers [23].

Mossin's analysis [17] based on intersection types is exact for the simply-typed λ-calculus with recursion under the full reduction semantics (i.e., β-reductions can be applied inside λ-abstractions). To our knowledge, his algorithm has never been implemented. Given a term e, his algorithm unfolds recursion a certain (huge) number of times to obtain a recursion-free (thus strongly normalizing) term e' that has the same flow set as e, and then fully reduces e' to obtain the flow set. Both the number of required unfoldings is huge, so that his algorithm would not be runnable even for the small benchmark programs in Section 6.

Vardoulakis and Shivers [26, 25], and Earl et al. [3] have recently proposed new control flow analyses, where programs are modeled as *first-order* pushdown systems. Our CFA based on HO model checking may have some connection to their methods, since we model programs (via the two-step encodings) as higher-order recursion schemes, which are equivalent to *higher-order* (collapsible) pushdown systems [5]. It would be interesting to consider something between our CFA (based on higher-order pushdown in the sense above) and theirs, like "CFA based on 2nd-order pushdown systems".

Ong and Tzevelekos [19] studied the (un)decidability of control flow analysis of *open* higher-order functional programs (i.e. programs that may have unknown arguments), and shown that the CFA problem in that setting is undecidable in general but that it is decidable for a certain fragment. In the present paper, we considered only closed programs.

Our CFA benefits from recent advances in higher-order model checking [18, 9, 7]. Kobayashi ([10], Section 3.3.2) sketched the reduction from CFA to HO model checking, but have neither formalized nor implemented it. Kobayashi et al. [9, 14, 13] have also applied higher-order model checking to other program analysis/verification problems for functional programs, and implemented tools for tree-processing programs [14] and reachability verification [13].

8 Conclusions

We have formalized a new method for control flow analysis based on HO model checking, and proved its correctness. It is exact for the simply-typed λ-calculus with recursion. We have also implemented the method and carried out preliminary experiments, to show that it is indeed runnable at least for small programs. We have to wait for further advances of HO model checking to judge its practicality, but we believe that the present work expands the design space for control flow analyses, by providing an analysis with the extreme precision.

Acknowledgments We would like to thank ananymous referees for useful comments. This work is partially supproted by Kakenhi 23220001 and 22·3842.

References

1. Blume, M., Acar, U.A., Chae, W.: Exception handlers as extensible cases. In: Proceedings of APLAS 2008. lncs, vol. 5356, pp. 273–289. springer (2008)
2. Danvy, O., Filinski, A.: Representing control: A study of the CPS transformation. Mathematical Structures in Computer Science 2(4), 361–391 (1992)
3. Earl, C., Might, M., Horn, D.V.: Pushdown control-flow analysis of higher-order programs. CoRR abs/1007.4268 (2010)
4. Fähndrich, M., Rehof, J.: Type-based flow analysis and context-free language reachability. Mathematical Structures in Computer Science 18(5), 823–894 (2008)
5. Hague, M., Murawski, A., Ong, C.H.L., Serre, O.: Collapsible pushdown automata and recursion schemes. In: Proceedings of 23rd Annual IEEE Symposium on Logic in Computer Science. pp. 452–461. IEEE Computer Society (2008)
6. Heintze, N., McAllester, D.: Linear-time subtransitive control flow analysis. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 261–272 (1997)
7. Kobayashi, N.: Model-checking higher-order functions. In: Proceedings of PPDP 2009. pp. 25–36. acm (2009)
8. Kobayashi, N.: TRecS: A type-based model checker for recursion schemes. http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/ (2009)

9. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proc. of POPL. pp. 416–428 (2009)
10. Kobayashi, N.: Model checking higher-order programs. Submitted for publication. A revised and extended version of [9] and [7], available at http://www.kb.ecei.tohoku.ac.jp/~koba/papers/hmc.pdf. (2010)
11. Kobayashi, N.: A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In: Proceedings of FoSSaCS 2011. lncs, vol. 6604, pp. 260–274. springer (2011)
12. Kobayashi, N., Ong, C.H.L.: Complexity of model checking recursion schemes for fragments of the modal mu-calculus. In: Proceedings of ICALP 2009. lncs, vol. 5556, pp. 223–234. Springer-Verlag (2009)
13. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proc. of PLDI (2011)
14. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: Proc. of POPL. pp. 495–508 (2010)
15. Lester, M.M., Neatherway, R.P., Ong, C.H.L., Ramsay, S.J.: THORS hammer. http://mjolnir.cs.ox.ac.uk/thors (2011)
16. Meyer, A.R., Wand, M.: Continuation semantics in typed lambda-calculi (summary). In: Logic of Programs. lncs, vol. 193, pp. 219–224. springer (1985)
17. Mossin, C.: Exact flow analysis. Mathematical Structures in Computer Science 13(1), 125–156 (2003)
18. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006. pp. 81–90. IEEE Computer Society Press (2006)
19. Ong, C.H.L., Tzevelekos, N.: Functional reachability. In: Proc. of LICS. pp. 286–295. IEEE Computer Society (2009)
20. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
21. Prabhu, T., Ramalingam, S., Might, M., Hall, M.W.: EigenCFA: accelerating flow analysis with gpus. In: Proc. of POPL. pp. 511–522 (2011)
22. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie-Mellon University (May 1991)
23. Shivers, O.: Higher-order control-flow analysis in retrospect: Lessons learned, lessons abandoned. In: ACM SiGPLAN Notices - Best of PLDI 1979-1999. pp. 257–269 (2003)
24. Tobita, Y., Tsukada, T., Kobayashi, N.: Exact flow analysis by higher-order model checking. An extended version of this paper, available from the 3rd author's home page (2012)
25. Vardoulakis, D., Shivers, O.: CFA2: a context-free approach to control-flow analysis. Logical Methods in Computer Science 7(2) (2011)
26. Vardoulakis, D., Shivers, O.: Pushdown flow analysis of first-class control. In: Proc. of ICFP. pp. 69–80. ACM Press (2011)
27. Wright, A.K., Jagannathan, S.: Polymorphic splitting: An effective polyvariant flow analysis. ACM Trans. Prog. Lang. Syst. 20(1), 166–207 (1998)