Information and Computation 211 (2012) 49–76

Contents lists available at SciVerse ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Control-flow analysis of function calls and returns by abstract interpretation

Jan Midtgaard [a,*], Thomas P. Jensen [b]

[a] Department of Computer Science, Aarhus University, Åbogade 34, DK-8200 Aarhus N, Denmark
[b] INRIA, Campus de Beaulieu, F-35042 Rennes, France

ARTICLE INFO

Article history:
Received 19 May 2010
Revised 2 November 2011
Available online 24 December 2011

Keywords:
Control-flow analysis
Abstract interpretation

ABSTRACT

Abstract interpretation techniques are used to derive a control-flow analysis for a simple higher-order functional language. The analysis approximates the interprocedural control-flow of both function calls and returns in the presence of first-class functions and tail-call optimization. In addition to an abstract environment, the analysis computes for each expression an abstract call-stack, effectively approximating where function calls return. The analysis is systematically derived by abstract interpretation of the stack-based C_aEK abstract machine of Flanagan et al. using a series of Galois connections. We prove that the analysis is equivalent to an analysis obtained by first transforming the program into continuation-passing style and then performing control flow analysis of the transformed program. We then show how the analysis induces an equivalent constraint-based formulation, thereby providing a rational reconstruction of a constraint-based CFA from abstract interpretation principles.

1. Introduction

Control-flow analysis (CFA) of functional programs is concerned with determining how the program's functions call each other. In the case of the lambda calculus, this amounts to computing the flow of lambda expressions in order to determine what functions are effectively called in an application $(e_1\ e_2)$. The result of a CFA can be visualized as an oriented *control flow graph* (CFG) linking sub-expression e_i to sub-expression e_j if evaluation of e_i may entail the immediate evaluation of e_j. A CFA computes an approximation of the actual behaviour of the program and can be more or less accurate depending on the technique employed.

In his seminal work, Jones [1,2] proposed to use program analysis techniques to statically approximate the flow of lambda-expressions under both call-by-value and call-by-name evaluation in the lambda calculus. Since then CFA has been the subject of an immense research effort [3–6]—see the recent survey by Midtgaard [7] for a complete list. CFA has been expressed using a variety of formalisms including data flow equations, type systems and constraint-based analysis. Surprisingly, nobody has employed Cousot's programme of *calculational abstract interpretation* [8] in which a program analysis is calculated by systematically applying abstraction functions to a formal programming language semantics. The purpose of this article is to show that such a derivation is indeed feasible and that a number of advantages follow from taking this approach:

* Corresponding author.
E-mail addresses: jmi@cs.au.dk (J. Midtgaard), Thomas.Jensen@inria.fr (T.P. Jensen).

0890-5401/$ – see front matter
doi:10.1016/j.ic.2011.11.005

- The systematic derivation of a CFA for a higher-order functional language from a well-known operational semantics provides the resulting analysis with strong mathematical foundations. Its correctness follows directly from the general theorems of abstract interpretation.
- The approach is easily adapted to different variants of the source language. We demonstrate this by deriving a CFA for functional programs written in continuation-passing style.
- The common framework of these analyses enables their comparison. We take advantage of this to settle a question about the equivalence between the analysis of programs in direct and continuation-passing style.
- The resulting equations can be given an equivalent constraint-based presentation, providing *ipso facto* a rational reconstruction and a correctness proof of constraint-based CFA.

The article is organized as follows. Section 2 provides a concise enumeration of fundamental notions from abstract interpretation used in the rest of the article. In Section 3 we define the language of study (the lambda calculus in administrative normal form) and its semantics, and give an example of CFA of programs written in this language. Sections 4 and 5 contain the derivation of a 0-CFA from an operational semantics: the C_aEK machine of Flanagan et al. [9]. In Section 4 we define a series of Galois connections that each specifies one aspect of the abstraction in the analysis. In Section 5 we calculate the analysis as the result of composing the collecting semantics induced by the abstract machine with these Galois connections. Section 6 uses the same technical machinery to derive a CFA for a language in continuation-passing style and sets up a relation between the two abstract domains that enables to prove a lock-step equivalence of the analysis of programs in direct style and the CPS analysis of their CPS counterparts. In Section 7 we show how the recursive equations defining the CFA of a program induce an equivalent formulation of the analysis, where the result of the analysis now is expressed as a solution to a set of constraints. Section 8 compares with related approaches and Section 9 concludes.

Preliminary versions of the results reported in this article were published at SAS 2008 [10] and ICFP 2009 [11]. The present article is a revised version of the latter paper, extending the lock-step relation between the direct and continuation-passing style analyses to include integer constants. The paper has furthermore been expanded with proofs and details of derivations of the abstract interpretations.

2. Abstract interpretation

This section recalls basic notions of lattice theory and abstract interpretation [12–16] on which we base our developments in the subsequent sections. In particular, we introduce the notion of *Galois connections* and provide a list of known Galois connections that will be used to design the abstraction underlying the CFA developed in Section 4.

A partially ordered set (poset) $\langle S;\sqsubseteq\rangle$ is a set S equipped with a partial order $\sqsubseteq$. A complete lattice is a poset $\langle C;\sqsubseteq,\bot,\top,\sqcup,\sqcap\rangle$, such that the least upper bound $\sqcup S$ and the greatest lower bound $\sqcap S$ exist for every subset S of C. $\bot=\sqcap C$ denotes the infimum of C and $\top=\sqcup C$ denotes the supremum of C. The set of total functions $D\to C$, whose codomain is a complete lattice $\langle C;\sqsubseteq,\bot,\top,\sqcup,\sqcap\rangle$, is itself a complete lattice $\langle D\to C;\dot{\sqsubseteq},\dot{\bot},\dot{\top},\dot{\sqcup},\dot{\sqcap}\rangle$ under the pointwise ordering $f\,\dot{\sqsubseteq}\,f'\Leftrightarrow\forall x.f(x)\sqsubseteq f'(x)$, and with bottom, top, join, and meet extended similarly. The powersets $\wp(S)$ of a set S ordered by set inclusion is a complete lattice $\langle\wp(S);\subseteq,\emptyset,S,\cup,\cap\rangle$.

2.1. Galois connections

A Galois connection is a pair of functions α, γ between two posets $\langle C;\sqsubseteq\rangle$ and $\langle A;\leqslant\rangle$ such that for all $a\in A$, $c\in C$: $\alpha(c)\leqslant a\Leftrightarrow c\sqsubseteq\gamma(a)$. Equivalently a Galois connection can be defined as a pair of functions satisfying:

(a) α and γ are monotone.
(b) $\alpha\circ\gamma$ is reductive (for all $a\in A$: $\alpha\circ\gamma(a)\leqslant a$).
(c) $\gamma\circ\alpha$ is extensive (for all $c\in C$: $c\sqsubseteq\gamma\circ\alpha(c)$).

Galois connections are typeset as $\langle C;\sqsubseteq\rangle\ \underset{\alpha}{\overset{\gamma}{\leftrightarrows}}\ \langle A;\leqslant\rangle$. We omit the orderings when they are clear from the context. For a Galois connection between two complete lattices $\langle C;\sqsubseteq,\bot_c,\top_c,\sqcup,\sqcap\rangle$ and $\langle A;\leqslant,\bot_a,\top_a,\vee,\wedge\rangle$, α is a complete join-morphism (CJM) (for all $S_c\subseteq C$: $\alpha(\sqcup S_c)=\vee\alpha(S_c)=\vee\{\alpha(c)\mid c\in S_c\}$) and γ is a complete meet morphism (for all $S_a\subseteq A$: $\gamma(\wedge S_a)=\sqcap\gamma(S_a)=\sqcap\{\gamma(a)\mid a\in S_a\}$). The composition of two Galois connections $\langle C;\sqsubseteq\rangle\ \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}}\ \langle B;\subseteq\rangle$ and $\langle B;\subseteq\rangle\ \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}}\ \langle A;\leqslant\rangle$ is itself a Galois connection $\langle C;\sqsubseteq\rangle\ \underset{\alpha_2\circ\alpha_1}{\overset{\gamma_1\circ\gamma_2}{\leftrightarrows}}\ \langle A;\leqslant\rangle$. Galois connections in which α is surjective (or equivalently γ is injective) are typeset as: $\langle C;\sqsubseteq\rangle\ \underset{\alpha}{\overset{\gamma}{\leftrightarrows}}\ \langle A;\leqslant\rangle$. Galois connections in which γ is surjective (or equivalently α is injective) are typeset as: $\langle C;\sqsubseteq\rangle\ \underset{\alpha}{\overset{\gamma}{\leftrightarrows}}\ \langle A;\leqslant\rangle$. When both α and γ are surjective, the two domains are isomorphic.

The following Galois connections will be used in the article: