Optimising Compilers - Notes on Constraint Based Analysis

Dominic Orchard

January 12, 2009

1 Introduction

Generally a *constraint-based analysis* traverses a program, emitting and solving constraints describing properties of the program. Many program analyses in essence solve a set of constraints e.g. constraining and unifying type information in Hindley-Milner type checking, or constraining the set of "live" variables in a statement with respect to successive statements in live-variable analysis. We consider here a constraint-based analysis of control-flow in functional programs.

Functional programming languages typically allow definition of higher-order functions – a function that takes a function as a parameter. Therefore control flow in functional languages is modified not only by conditional constructs (if statements), and standard function applications, but also by the application of functions passed as parameters.

The programming model behind functional languages is usually very different to the architectural model of the target machine, which is typically imperative. Knowing control-flow information for a functional program is useful for efficient compilation that makes use of imperative loop constructs and other optimisations such as inlining, tail-recursion, and specialisation – where the compiler can generate faster implementations of a function given information about parameters, types, and return values.

A brief note on terminology: A function definition has *formal parameters* e.g. $\lambda x. \ldots$ has formal parameter x. A function is called with (applied to) *actual parameters* e.g. $(\lambda x. \ldots)\ 2$ – the lambda expression has the actual parameter 2.

2 0-CFA [3, 4]

0-CFA (0th-order Control-Flow Analysis) is one such analysis of control-flow in functional programs. The "0th-order" naming denotes that contextual information is not considered; all local semantics are combined into a global

semantics. Although 0-CFA is a control-flow analysis the constraints are generated over a subset of a program's data-flow: the flow of *function values* i.e. λ expressions. Other values, such as integers, are not considered in the constraints although it is simple to extend 0-CFA to include the flow of other values through a program (as in the lecture notes) thus creating a mix of control-flow and data-flow analysis.

We consider 0-CFA on the following simple language where x ranges over variables, c ranges over integer constants, and $\oplus$ is a binary operation:

$$e ::= x \mid c \mid \lambda x.e \mid e_1\, e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid e_1 \oplus e_2$$

In 0-CFA each expression and subexpression in a program is labelled with a unique integer i and associated with the *flow* set α_i of function values that can be returned by (or can "flow" from) evaluation of the subexpression. Subset constraints are generated on the flow sets for an expression and its subexpressions. The constraints can be satisfied iteratively until a fixed-point is reached. Initially all flow sets $\alpha_i = \emptyset$. The rules for constraint generation are as follows:

[const]	c^i	$\rightarrow$	no constraints
[op]	$(e_1^j \oplus e_2^k)^i$	$\rightarrow$	no constraints
[var]	x^i	$\rightarrow$	$\alpha_j \subseteq \alpha_i$ where x^j is the binding of x
[lam]	$(\lambda x^j.e^k)^i$	$\rightarrow$	$\{(\lambda x^j.e^k)\} \subseteq \alpha_i$
[app]	$(e_1^j e_2^k)^i$	$\rightarrow$	$\forall(\lambda x^a.e^b) \in \alpha_j : (\alpha_k \subseteq \alpha_a \wedge \alpha_b \subseteq \alpha_i)$
[let]	$(\texttt{let}\ x^l\ \texttt{=}\ e_1^j\ \texttt{in}\ e_2^k)^i$	$\rightarrow$	$\alpha_k \subseteq \alpha_i \wedge \alpha_j \subseteq \alpha_l$
[if]	$(\texttt{if}\ e_1^j\ \texttt{then}\ e_2^k\ \texttt{else}\ e_3^l)^i$	$\rightarrow$	$\alpha_k \subseteq \alpha_i \wedge \alpha_l \subseteq \alpha_i$

The majority of the rules are straightforward.

- [const] and [op]: Neither have associated flow sets or constraints as neither can return a function value.
- [var]: For a variable reference, labelled i, there must be a binding of the variable, labelled j, either as a `let` expression or as the formal parameter of a function e.g.

  $$\texttt{let}\ x^j\ \texttt{=}\ \ldots\ \texttt{in}\ \ldots x^i \ldots$$

  or

  $$\lambda x^j \ldots x^i \ldots$$

  The generated constraint $\alpha_j \subseteq \alpha_i$ ensures that any function values bound to x^j are in the flow set for x^i.
- [lam]: The defined function value is in the flow set α_i.

- [let]: The set α_k for the `let` body is a subset of the set for the whole expression α_i. The set of the bound expression α_j is a subset of the set for the target variable of the binding α_l.
- [if]: The set for the "true" branch and the "false" branch are subsets of the flow set for the overall expression. The constraint is equivalent to $\alpha_k \cup \alpha_l \subseteq \alpha_i$ but is defined without a union for consistency.

The function application rule is more interesting and is described in depth below with an example.

- [app] $(e_1^j e_2^k)^i \rightarrow \forall(\lambda x^a.e^b) \in \alpha_j : (\alpha_k \subseteq \alpha_a \wedge \alpha_b \subseteq \alpha_i)$

  The rule matches an application $(e_1^j e_2^k)^i$, where e_1^j is an expression returning a function, and generates constraints for each possible function that may be returned by e_1^j e.g. e_1^j may be a variable bound to any of a number of functions.

  For each function in the flow set for e_1^j, i.e. $\forall(\lambda x^a.e^b) \in \alpha_j$, the following constraints are generated:

  - $\alpha_k \subseteq \alpha_a$, relating to the formal parameters of the applied function: The flow set of the *actual* parameter to the function, labelled k, is a subset of the flow set of the *formal* parameter, labelled a. Therefore the flow set α_a for the formal parameter of a function contains all actual parameters from the various applications of the function in the program (thus realising the global semantics of 0-CFA, cf. k-CFA, section 2.1).
  - $\alpha_b \subseteq \alpha_i$, relating to the return result of the applied function: The return values of the function, flow set α_b, are a subset of the return values of the application expression, α_i. Therefore the flow set of the application expression α_i contains all return flow sets of all possibly applied functions.

  Therefore the [app] rule provides constraints on how function values flow to formal parameters, are applied, and returned, hence facilitating an analysis of function application, and thus control-flow, within a functional program.

  Consider the following example code and a labelling:

$$\begin{aligned} f &= \lambda x.x * 3 \\ g &= \lambda x.x + 2 \\ h &= \lambda x.x\ 2 \\ z &= h\ f\ +\ h\ g \end{aligned}$$

(a) unlabelled

$$\begin{aligned} f &= (\lambda x^2.(x * 3)^3)^1 \\ g &= (\lambda x^5.(x + 2)^6)^4 \\ h &= (\lambda x^8.(x^9\ 2^{10})^{11})^7 \\ z &= (h^{13}\ f^{14})^{12}\ +\ (h^{16}\ g^{17})^{15} \end{aligned}$$

(b) partially labelled

The function h takes a function as a parameter, bound to the formal parameter x, hence h is a high-order function. The function bound to x is then applied to the number 2.

The [app] rule applied to the two applications of h in the definition for z produces $\{(\lambda x^2.(x*3)^3)^1, (\lambda x^5.(x+2)^6)^4\} \subseteq \alpha_9$. The derivation of this is not shown for brevity but it should be clear that the formal parameter for h can only ever be the f lambda expression or the g lambda expression.

Consider the [app] rule applied to the function body of h:

$$[\text{app}] \quad (x^9\ 2^{10})^{11} \quad \rightarrow \quad \forall(\lambda x^a.e^b) \in \alpha_9 : (\alpha_{10} \subseteq \alpha_a \wedge \alpha_b \subseteq \alpha_{11})$$

Now $\forall(\lambda x^a.e^b) \in \alpha_9$:

- $(\lambda x^2.(x*3)^3)^1 \in \alpha_9$ generates: $\alpha_{10} \subseteq \alpha_2 \wedge \alpha_3 \subseteq \alpha_{11}$.
- $(\lambda x^5.(x+2)^6)^4 \in \alpha_9$ generates: $\alpha_{10} \subseteq \alpha_5 \wedge \alpha_6 \subseteq \alpha_{11}$.

As neither f nor g take a function or return a function the flow sets for their parameters and return values are all empty. To make the relationship clearer for this example we temporarily assume that constraints are generated for all integer values, therefore adding further data-flow information into the analysis, via these rules:

$$\begin{array}{lrcl} [\text{const}] & c^i & \rightarrow & \{c\} \subseteq \alpha_i \\ [\text{op}] & (e_1^j \oplus e_2^k)^i & \rightarrow & \{\alpha^j \oplus \alpha^k\} \subseteq \alpha_i \end{array}$$

Applying the integer constraints to the example and solving the constraints on the functions produces the following flow sets:

$\{2\} \subseteq \alpha_{10}$	2 is the only actual parameter to the application in h
$\alpha_{10} \subseteq \alpha_2 \Rightarrow \{2\} \subseteq \alpha_2$	2 flows to the formal parameter for f
$\alpha_{10} \subseteq \alpha_5 \Rightarrow \{2\} \subseteq \alpha_5$	2 flows to the formal parameter for g
$\alpha_3 \subseteq \alpha_{11} \Rightarrow \{3*2\} \subseteq \alpha_{11}$	3*2 is a possible return result of the application in h
$\alpha_6 \subseteq \alpha_{11} \Rightarrow \{2+2\} \subseteq \alpha_{11}$	2+2 is a possible return result of the application in h
	$\Rightarrow \{3*2, 2+2\} \subseteq \alpha_{11}$

For further reading on CFA see [3, 2, 4, 5]. The usual presentation of 0-CFA is as a function from labels to term constraints, a separate function from variable names to variable constraints, and inference rules for the derivation of such constraints from terms in a language. The function of variable names to variable constraints does not differentiate between variables in different scopes. Separation of constraints for variables of the same name in different scopes was provided in these notes by generating constraints on flow sets for all terms and variables together, indexed by unique labels.

2.1 k-CFA

A quick note on k-CFA [non-examinable]: k-CFA takes into account scoping and the context of function application thus differentiating between the different points of application (or call sites) of a function. 0-CFA collects all actual parameter flow sets, from all application points, into the formal parameter flow set; 0-CFA returns no information on static or dynamic point of application. k-CFA has sequentially labelled flow sets for each successive application of a function. Therefore, for example, k-CFA can differentiate between the first and subsequent applications of a recursive function. Further function specialisation and inlining can be achieved given more precise information about a function's application at specific program points.

2.2 Relation to the 2008 Lecture Notes

These notes provided clarification of the CFA material [1], presenting rules for generating constraints that differ slightly in presentation from those given in the lecture notes. For function application these notes presented the rule for generating constraints as:

$$[\text{app}]\ (e_1^j e_2^k)^i \rightarrow \forall(\lambda x^a.e^b) \in \alpha_j : (\alpha_k \subseteq \alpha_a \wedge \alpha_b \subseteq \alpha_i) \tag{1}$$

On the other hand, the 2008 lecture notes present the following rule (page 27):

$$(e_1^j e_2^k)^i \rightarrow \quad (\alpha_k \mapsto \alpha_i) \supseteq \alpha_j \tag{2}$$

which is shorthand for further generated constraints:

$$\text{whenever } \alpha_j \supseteq \{(\lambda x^q.e^r)^p\} \rightarrow \alpha_q \supseteq \alpha_k \wedge \alpha_i \supseteq \alpha_r \tag{3}$$

The generated constraints of (3) are equivalent to (1) presented in these notes: "whenever $\alpha_j \supseteq \{\lambda \ldots$" in (3) is equivalent to "$\forall(\lambda \ldots) \in \alpha_j$" in (1) and the following clause in (3) is written using $\supseteq$ as opposed to $\subseteq$ in (1) – which was chosen for clairty when left-to-write reading.

In the example given in the lecture slides the short hand constraint (2) is seen in a general form on slide 19 of lecture 11, and is instantiated on slide 20. These shorthand constraints are encountered in the example on slides 28 and 32 and are expanded to generate two constraints (each clause of the $\wedge$ in (3)) on slide 29 and 33 respectively (show in blue on the slides).

Additionally in the lecture slides the example takes into account integer values in the flow sets for expressions – hence the analysis is actually a mix of control and data-flow analysis.

References

[1] Optimising Compilers, Cambridge, Computer Laboratory, 2008. `http://www.cl.cam.ac.uk/teaching/0809/OptComp/`.

[2] Nevin Heintze. Set-based analysis of ML programs. *SIGPLAN Lisp Pointers*, VII(3):306–317, 1994.

[3] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[4] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 332–345, New York, NY, USA, 1997. ACM.

[5] O. Shivers. Control flow analysis in Scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM.