Minimal function graphs are not instrumented

Alan Mycroft* & Mads Rosendahl†

* *Computer Laboratory, Cambridge University*
New Museums Site, Pembroke Street, Cambridge CB2 3QG, United Kingdom
E-mail: `Alan.Mycroft@cl.cam.ac.uk`
† *DIKU, Copenhagen University*
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
E-mail: `rose@diku.dk`

Résumé: The minimal function graph semantics of Jones and Mycroft is a standard denotational semantics modified to include only 'reachable' parts of a program. We show that it may be expressed directly in terms of the standard semantics without the need for instrumentation at the expression level and, in doing so, bring out a connection with strictness. This also makes it possible to prove a stronger theorem of correctness for the minimal function graph semantics.

Keywords: Minimal function graph, strictness, neededness, abstract interpretation.

Commonly, in abstract interpretation, we start with a *standard semantics* in denotational style, modify this to make an *instrumented semantics* which includes further operational detail (for example by adding timing information) and finally *abstract* this to form a decidable *abstract semantics.* Much work has gone into the theory of the abstraction process, but relatively little into the instrumentation process. We note that choices (or errors) in the instrumentation can lead to very different conclusions about execution. For example, if we instrument a semantics by replacing values v with pairs (v, n) where n is the number of steps required to reduce an expression e to v then we have possible sensible representations for the instrumented version of subtraction as $\lambda((v,n),(v',n')).(v-v', n+n'+1)$ (sequential) and $\lambda((v,n),(v',n')).(v-v', max(n,n')+1)$ (parallel) together with less reasonable definitions such as $\lambda((v,n),(v',n')).(v-v', 42)$.

The *minimal function graph* semantics was defined by Jones and Mycroft [5] and in part represents a special case of the Cousots' work [3] on analysing recursive procedures by predicate transformers. It is specified as an instrumented semantics leaving to intuition questions about whether

A fuller version of this paper appears as [7].

the instrumentation was appropriate. We sidestep this issue by showing that, with a slight modification, the minimal function graph semantics can be defined in terms of the standard semantics. We also show that there are two subtly different MFG semantics according to how the definition of reachability is interpreted — either as strictness or as neededness.

1. First-order Language

We consider a language of first order recursion equations (of n functions f, each of arity k) with parameters x, constants c, conditional expressions, base functions a (presumed arity k) and defined functions f (also arity k). Expressions e follow a usual first-order syntax. Programs p (also considered as declarations) have the form

$$\begin{aligned} &f_1(x_1,\ldots,x_k) \;= e_1; \\ &\vdots \\ &f_n(x_1,\ldots,x_k) \;= e_n. \end{aligned}$$

The meaning of a program is an n-tuple of functional values. Free (variable) identifiers in e_i $(1 \le i \le n)$ are a subset of $\{x_1,\ldots,x_k\}$ and free function identifiers a subset of $\{f_1,\ldots,f_n\}$.

Standard semantics. This work only considers call-by-value semantics.[1] A semantics requires a set of *ground values* V (we assume $\mathrm{Z} \subseteq V$ and $\mathrm{B} \subseteq V$ and moreover that $\perp \notin V$ so lifting: $V \to V_\perp$ can be treated as identity). It further requires an interpretation for each constant c_j, having as meaning $const_j \in V_\perp$, and each basic operation a_j, having as meaning $basic_j : V^k \to V_\perp$. In examples we will feel free to use other names for constants, basic operations, parameters and functions, particularly $\perp$ for an expression whose value is $\perp$.

The semantics below assumes a conditional function "$\cdot \to \cdot | \cdot$" and a function to enforce call-by-value:

$$\begin{aligned} &\mathit{strictapp} : (V^k \to V_\perp) \to (V_\perp)^k \to V_\perp \\ &\mathit{strictapp}\; f\; (v_1,\ldots,v_k) = f(v_1,\ldots,v_k) \quad \text{if all } v_i \ne \perp \\ &\qquad\qquad\qquad\qquad\qquad\;\; = \perp \qquad\qquad\qquad \text{otherwise.} \end{aligned}$$

[1]A companion paper under construction extends this framework to the lazy case.

Semantic domains and functions

$$
\begin{array}{lll}
V & & \text{values, as above} \\
\mathit{ValEnv} & = V^k & \\
\mathit{FnEnv} & = (V^k \to V_\perp)^n &
\end{array}
\qquad
\begin{array}{l}
\mathbf{E}\,[\![e]\!] : \mathit{FnEnv} \to \mathit{ValEnv} \to V_\perp \\
\mathbf{D}\,[\![p]\!] : \mathit{FnEnv} \to \mathit{FnEnv} \\
\mathbf{P}\,[\![p]\!] : \mathit{FnEnv}
\end{array}
$$

Semantic equations

$$
\begin{array}{ll}
\mathbf{E}\,[\![x_i]\!]\eta\rho & = \rho_i \\
\mathbf{E}\,[\![c_j]\!]\eta\rho & = \mathit{const}_j \\
\mathbf{E}\,[\![\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3]\!]\eta\rho & = \mathbf{E}\,[\![e_1]\!]\eta\rho \to \mathbf{E}\,[\![e_2]\!]\eta\rho \mid \mathbf{E}\,[\![e_3]\!]\eta\rho \\
\mathbf{E}\,[\![a_j(e_1,\ldots,e_k)]\!]\eta\rho & = \mathit{strictapp}\ \mathit{basic}_j\ (\mathbf{E}\,[\![e_1]\!]\eta\rho,\ldots,\mathbf{E}\,[\![e_k]\!]\eta\rho) \\
\mathbf{E}\,[\![f_j(e_1,\ldots,e_k)]\!]\eta\rho & = \mathit{strictapp}\ \eta_j\ (\mathbf{E}\,[\![e_1]\!]\eta\rho,\ldots,\mathbf{E}\,[\![e_k]\!]\eta\rho) \\
\multicolumn{2}{l}{\mathbf{D}\,[\![f_1(x_1\ldots x_k) = e_1;\ldots;f_n(x_1\ldots x_k) = e_n]\!] = \lambda\eta.\langle \mathbf{E}\,[\![e_1]\!]\eta,\ldots,\mathbf{E}\,[\![e_n]\!]\eta\rangle} \\
\multicolumn{2}{l}{\mathbf{P}\,[\![p]\!] = \mathit{fix}\ \mathbf{D}\,[\![p]\!]}
\end{array}
$$

2. Strictness and neededness of a sub-expression

Recall that, in a lazy semantics, an expression e is strict in a (free) variable x if, whenever x is bound to a non-terminating expression then e fails to terminate. Similarly informally e needs x if reduction of e requires reduction of whatever expression x is bound to. In general neededness implies strictness, but the implication is proper — consider an expression such as "$\perp$" which is strict in all variables but does not need any variable. We now turn to the concepts of strictness and neededness of sub-expressions which are sensible even in call-by-value languages such as ours.

Strictness. We say that an expression $e[e']$ is *strict in* the subexpression e' if, whenever e' fails to terminate then so does $e[e']$. This can be characterised in terms of the standard semantics as $\mathbf{E}\,[\![e[\perp]]\!]\eta\rho = \perp$.

Abusing notation a little, given $\eta \in \mathit{FnEnv} = (V^k \to V_\perp)^n$ we define updating of one component of η by:

$$\eta[(i,\vec{v}) \mapsto w] = \langle \eta_1,\ldots,\eta_{i-1},\eta_i[\vec{v} \mapsto w],\eta_{i+1},\ldots,\eta_n\rangle.$$

Now, suppose f is a function in a program p, for example

$$f(x) = \mathit{if}\, x = 0\ \mathit{then}\ 1\ \mathit{else}\ x * f(x-1)$$

and $\vec{v} = (v_1, \ldots, v_k)$ is a tuple of argument values. We say e is *strict* in $f(\vec{v})$ if e fails to terminate under the assumption that $f(\vec{v})$ fails to terminate. This has two possible meanings according to whether we wish to consider (intensionally) the internal calls, so $f(7)$ is strict in $f(7), f(6), \ldots, f(0)$, or (extensionally) the function as a black box, so $f(7)$ is just strict in itself. In terms of $\mathbf{E}\,[\![\cdot]\!]$ the two choices can be written as

$$\begin{array}{ll} \mathbf{E}\,[\![e]\!](\mathit{fix}\ \lambda\eta.((\mathbf{D}\,[\![p]\!]\eta)[(1,\vec{v}) \mapsto \bot]))\rho = \bot & \text{intensional} \\ \mathbf{E}\,[\![e]\!](\mathit{fix}\ \mathbf{D}\,[\![p]\!])[(1,\vec{v}) \mapsto \bot]\rho = \bot & \text{extensional} \end{array}$$

We select the first, intensional, meaning — this corresponds to the intuitive motivation above and is formalised by the $\mathbf{S}\,[\![\cdot]\!]$ semantic function below.

Neededness. Above we said that an expression e is strict in a subexpression e' if, whenever e' is a non-terminating expression, then e fails to terminate. We similarly say that e *needs* e' if under the (same) assumed operational semantics, reduction of e requires reduction of e'. For such an operational definition, strictness and neededness only differ subtly — they agree in all circumstances except that an expression which never terminates is always strict in any subexpression but may not *need* a given subexpression. Barendregt et al. [1] discuss neededness in the λ-calculus in some detail.

When defining neededness formally the problem becomes a little more subtle in that evaluation order which is invisible in the standard semantics may become visible in a neededness semantics.[2] Typically, however, we wish to abstract from such operational detail and we follow the published version of [5]. It takes the view that the needed semantics consists of those function/argument pairs which are needed by *some* operational semantics consistent with the prescribed denotational semantics.

3. Formal strictness and neededness

Strict or needed calls are represented as sets of argument tuples associated with each function name. We will use the set C given by

$$C = \mathcal{P}(\{1, \ldots, n\} \times V^k)$$

[2]Consider the contrived example
$$\begin{array}{ll} \mathit{main}() & = h(f(1), g(2)) \\ f(x) & = f(x) \\ g(y) & = g(y) \\ h(x,y) & = x + y \end{array}$$

where given $c \in C$ a pair $\langle i, \vec{v} \rangle \in c$ indicates that the i^{th} function f_i will be strict in (respectively need) the argument tuple $\vec{v}$.

Strictness semantics, $\mathbf{S}[\![\cdot]\!]$. Recalling the function-update notation from section 2 and given 'external' calls $c \in C$, the strict calls in p are

$$\mathbf{S}[\![p]\!]c = \bigcup_{\langle j,\vec{x}\rangle \in c} \{\langle i, \vec{v}\rangle \mid ((\mathit{fix}\lambda\eta.((\mathbf{D}[\![p]\!]\eta)[(i,\vec{v}) \mapsto \bot])) \downarrow j)(\vec{x}) = \bot\}$$

One should note that $\mathbf{S}$ is defined from $\mathbf{D}$ alone and does not use the actual expressions in function definitions.

Neededness semantics, $\mathbf{N}[\![\cdot]\!]$. We may define a neededness function $\mathbf{N}$ which is very similar to $\mathbf{S}$ with the intuition

$$\begin{aligned}\mathbf{N}[\![p]\!]c = \{\langle i, \vec{v}\rangle \mid \exists \langle j, \vec{w}\rangle \in c.\ & \textit{evaluating } f_j(\vec{w}) \textit{ necessitates}\\ & \textit{the evaluation of } f_i(\vec{v})\}\end{aligned}$$

This rather informal definition may also be found in [3]. Due credit is not always given to the Cousots for the idea of neededness. Their language also sidestepped the problem of argument evaluation order apparent in the example in section 2 by requiring the programmer to serialise argument evaluation.

The above informal requirement can be formalised *via* Jones and Mycroft's instrumented semantics which replaces values by pairs of values and function calls encountered during their evaluation. For our purposes it is convenient to write a separate semantic function $\mathbf{C}[\![\cdot]\!]$ yielding the second component of their pair as an adjunct to defining $\mathbf{N}[\![\cdot]\!]$:

$$\begin{aligned}
\mathbf{C}[\![x_j]\!]\eta\rho &= \{\}\\
\mathbf{C}[\![c_j]\!]\eta\rho &= \{\}\\
\mathbf{C}[\![\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3]\!]\eta\rho &= \mathbf{C}[\![e_1]\!]\eta\rho \cup (\mathbf{E}[\![e_1]\!]\eta\rho \rightarrow \mathbf{C}[\![e_2]\!]\eta\rho \mid \mathbf{C}[\![e_3]\!]\eta\rho)\\
\mathbf{C}[\![a_j(e_1,\ldots,e_k)]\!]\eta\rho &= \mathbf{C}[\![e_1]\!]\eta\rho \cup \cdots \cup \mathbf{C}[\![e_k]\!]\eta\rho\\
\mathbf{C}[\![f_j(e_1,\ldots,e_k)]\!]\eta\rho &= \mathbf{C}[\![e_1]\!]\eta\rho \cup \cdots \cup \mathbf{C}[\![e_k]\!]\eta\rho \cup\\
&\quad\ \{\langle i, (\mathbf{E}[\![e_1]\!]\eta\rho, \ldots, \mathbf{E}[\![e_k]\!]\eta\rho)\rangle\}
\end{aligned}$$

$$\mathbf{N}[\![p]\!]c = \mathit{fix}\left(\lambda b \in C.c \cup \bigcup_{\langle i,\vec{v}\rangle \in b} \mathbf{C}[\![e_i]\!](\mathbf{P}[\![p]\!])\vec{v}\right)$$

Minimal neededness. $\mathsf{N}[\![\cdot]\!]$ gave a 'maximal' neededness interpretation in which a variable is needed if *some* compatible interpreter needs it. There is also a 'minimal' neededness interpretation semantics which helps to identify the difference between S and N *via* a sandwich construction.

$$\mathsf{M}[\![p]\!]c = \bigcup_{\langle j,\vec{x}\rangle \in c} \{\langle i,\vec{v}\rangle \mid ((\mathit{fix}\lambda\eta.((\mathsf{D}[\![p]\!]\eta)[(i,\vec{v}) \mapsto \bot])) \downarrow j)(\vec{x}) = \bot \\ \wedge (\mathsf{P}[\![p]\!] \downarrow j)(\vec{x}) \neq \bot\}$$

Notice that $\mathsf{M}[\![\cdot]\!]$ follows $\mathsf{S}[\![\cdot]\!]$ in only using the expressions in a program *via* their standard denotations.

Sandwich lemma. For all programs p, we have $\forall c \in C.\, \mathsf{M}[\![p]\!]c \subseteq \mathsf{N}[\![p]\!]c \subseteq \mathsf{S}[\![p]\!]c$.

The program $f(x) = f(x)$ shows that all inclusions may be proper: $\mathsf{S}[\![p]\!]\{42\} = V$, $\mathsf{N}[\![p]\!]\{42\} = \{42\}$ and $\mathsf{M}[\![p]\!]\{42\} = \{\}$.

4. Minimal function graphs

From either the needed calls N or the strict calls S we may define MFG semantics as semantic functions MFG_X (with X ranging over $\{\mathsf{N},\mathsf{S},\mathsf{M}\}$).

First we need to define a notion of the space of (partial) function graphs G, ranged over by g. This is given by

$$G = \mathcal{P}(\{1,\ldots,n\} \times V^k \times V_\bot)$$

Jones and Mycroft used the space G' given by

$$G' = (V^k \to (V_\bot)_\bot)^n \cong \{1,\ldots,n\} \times V^k \to (V_\bot)_\bot$$

which has an obvious natural embedding in G. Our semantics only uses values in the embedding of G' within G.

The three MFG semantics are mappings from 'initial' or 'external world' calls (in C) to reachable parts of the function graph (in G). It is convenient to define first a function graph semantics

$$\mathsf{FG}[\![p]\!] : C \to G$$
$$\mathsf{FG}[\![p]\!]c = \{\langle i,\vec{v},r\rangle \mid \langle i,\vec{v}\rangle \in c \wedge (\mathsf{P}[\![p]\!] \downarrow i)(\vec{v}) = r\}.$$

The MFG semantics are then simply given by

$$\mathbf{MFG}_{\mathsf{X}}\,[\![p]\!] : C \to G$$
$$\mathbf{MFG}_{\mathsf{X}}\,[\![p]\!]c = \mathbf{FG}\,[\![p]\!](\mathbf{X}\,[\![p]\!]c).$$

The rest of this section discusses the relationship between the standard semantics $\mathbf{P}$ and the minimal function graph semantics $\mathbf{MFG}_{\mathsf{N}}$ and $\mathbf{MFG}_{\mathsf{S}}$.

Correctness. Unlike Jones and Mycroft's formulation the $\mathbf{MFG}_{\mathsf{S}}$ semantics is defined directly from the standard semantics. This makes it much easier to verify various connections between these semantics. The proofs may be found in the full version of the paper.

• Any non-$\perp$ result in an argument-result pair obtained by the minimal function graph interpretation can also be obtained by the standard semantics:

$$\forall c \in C. \langle i, \vec{v}, r\rangle \in \mathbf{MFG}_{\mathsf{S}}\,[\![p]\!]c \wedge r \neq \perp \Rightarrow r = (\mathbf{P}\,[\![p]\!] \downarrow i)(\vec{v})$$

This is the only proof included in [5]. In this case it follows trivially from the definition of $\mathbf{MFG}_{\mathsf{S}}$. A similar result may be proved for the $\mathbf{MFG}_{\mathsf{N}}$ semantics.

• Any non-$\perp$ result produced by the standard semantics can also be produced by the minimal function graph interpretation when given the arguments to the call as the specification of initial calls:

$$(\mathbf{P}\,[\![p]\!] \downarrow i)(\vec{v}) = r \wedge r \neq \perp \Rightarrow \langle i, \vec{v}, r\rangle \in \mathbf{MFG}_{\mathsf{S}}\,[\![p]\!]\{\langle i, \vec{v}\rangle\}$$

This relationship was not stated or proved in [5]. For the $\mathbf{MFG}_{\mathsf{S}}$ semantics it follows directly from the definition. A similar result may be proved for the $\mathbf{MFG}_{\mathsf{N}}$ semantics.

• We have claimed that the semantics $\mathbf{MFG}_{\mathsf{N}}$ is identical to the minimal function graph semantics in [5]. Although this is intuitively quite obvious the proof is not trivial as the needs in [5] are computed from a function environment which only contain information about previously called functions. A proof of this may be based on chaotic iteration sequences [3].

Sandwich theorem (from sandwich lemma).

$$\forall c \in C.\ \mathbf{MFG}_{\mathsf{M}}\,[\![p]\!]c \subseteq \mathbf{MFG}_{\mathsf{N}}\,[\![p]\!]c \subseteq \mathbf{MFG}_{\mathsf{S}}\,[\![p]\!]c.$$

5. Conclusion

We have shown how the minimum function graph semantics can be defined denotationally which facilitates statements and proofs of correctness. The process links the MFG semantics with the notions of strictness and neededness of subexpressions.

Acknowledgment. This research was supported by UK SERC grant GR/H14465 and by the Danish SNF 'DART' grant. We wish to thank Patrick and Radhia Cousot for their comments on a draft version of this paper.

References

[1] H. P Barendregt, J. R Kennaway, J. W Klop and M. R Sleep. *Needed Reduction and Spine Strategies for the Lambda Calculus.* In *Information and Control*, vol. 73, 1987.

[2] P Cousot and R Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In *4th POPL, Los Angeles, CA*, pp. 238–252, Jan., 1977.

[3] P Cousot and R Cousot. *Static determination of dynamic properties of recursive procedures.* In *Formal Description of Programming Concepts* (E. J Neuhold, ed.). North-Holland, 1978.

[4] P Cousot and R Cousot. *Inductive definitions, semantics and abstract interpretation* In *19th POPL*, Jan., 1992.

[5] N. D Jones and A Mycroft. *Data Flow Analysis of Applicative Programs using Minimal Function Graphs.* In *13th POPL, St. Petersburg, Florida*, pp. 296–306, Jan., 1986.

[6] A Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. Thesis. Univ. of Edinburgh, Dec., 1981.

[7] A Mycroft and M Rosendahl. *Minimal function graps are not instrumented.* Computer Laboratory technical report. Univ. of Cambridge, 1992.