Higher-Order Minimal Functional Graphs.

Conference Paper · September 1994

DOI: 10.1007/3-540-58431-5_17 · Source: DBLP

2 authors:

Neil D. Jones
University of Copenhagen
151 PUBLICATIONS **7,548** CITATIONS

SEE PROFILE

Mads Rosendahl
Roskilde University
8 PUBLICATIONS **149** CITATIONS

SEE PROFILE

Higher-Order Minimal Function Graphs

Neil D Jones[1] and Mads Rosendahl[2]
[1]Datalogisk Institut, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø
Denmark
[2]University of Roskilde, Dept of Computer Science
DK-4000 Roskilde, PO Box 260, House 20.2
Denmark
e-mail: neil@diku.dk[1] and madsr@dat.ruc.dk[2]

September 11, 1997

Abstract

We present a minimal function graph semantics for a higher-order functional language with applicative evaluation order. The approach extends previous results on minimal function graphs to higher-order functions. The semantics captures the intermediate calls performed during the evaluation of a program. This information may be used in abstract interpretation as a basis for proving the soundness of program analyses. An example of this is the "closure analysis" of partial evaluation.

Program flow analysis is concerned with obtaining an approximate but safe description of a program's run-time behaviour without actually having to run it on its (usually infinite) input data set.

Consider a functional program. The meaning of a simple function definition

$$f(x_1, \ldots, x_k) = e$$

is typically given by a semantic function $\mathbf{M}$ of type

$$\mathbf{M}[\![f(x_1, \ldots, x_k) = e]\!] : V^k \to V_\perp$$

where V is a set of values. Denotational semantics traditionally proceeds by defining $\mathbf{M}[\![p]\!]$ by recursion on syntax. Semantics-based program approximation can be done naturally by approximating a function on precise values by another function defined on abstract values (eg. $\perp, \top$). This approach is taken in [2] for strictness analysis and may be used for other *compositional* analyses where program parts can be analysed independently of the contexts in which they occur.

In contrast, a *top-down* analysis has as goal to describe the effects of applying the program to a given set of input values and so is context-dependent. This type of analysis cannot easily be related to a usual denotational semantics without some extra instrumentation. A well-known alternative is to describe functions by their graphs. It is natural to think of function f as a set of input-output pairs $IO_f = \{(a, f(a)), (b, f(b)), \ldots\}$, so a set of needed function values corresponds to a perhaps proper subset of IO_f. This leads to an approach described in [4]. In the minimal function graph approach to semantics the idea is to describe functions by the sets of argument-result pairs sufficient to identify the function as used. From a given argument tuple the minimal function graph description should give the smallest set of argument-result pairs that is needed to compute the result. As a semantic function it may have the type

$$\mathbf{M}[\![f(x_1, \ldots, x_k) = e]\!] : V^k \to \mathcal{P}(V^k \times V_\perp)$$

and the informal definition may be

$$\mathbf{M}[\![f(x_1, \ldots, x_k) = e]\!]\vec{v} = \{\langle \vec{u}, r\rangle \mid f(\vec{u}) = r \wedge f(\vec{v}) needs f(\vec{u})\}$$

By this approach one analyses a program by approximating its MFG semantics, as outlined in [4] and developed in several places since.

The stating point of this paper is: how can one approximate the behaviour of higher-order programs using the minimal function graph approach of only describing reachable values?

Our approach is again semantically based, but on a more operational level than for example [2]. Three steps are involved.

A Define a closure-based semantics for higher-order programs (in a now traditional way).

B Define a minimal function graph variant of this, which collects only reachable function arguments and results.

C Verify safety of program analyses by reference to this higher-order MFG semantics.

In this paper we present A and B in some detail and sketch an application as in C.

1 Language

Consider a small language based on recursion equation systems. The language allows higher-order functions and the evaluation order is eager. A program is a number of function definitions:

$$\begin{array}{ll} f_1 & x_1 \cdots x_k = e_1 \\ \vdots & \vdots \\ f_n & x_1 \cdots x_k = e_n \end{array}$$

An expression is built from parameters and function names by application and basic operations.

x_i	Parameters
f_i	Function names
$a_i(e_1, \ldots, e_k)$	Basic operations
$e_1(e_2)$	Application
if e_1 **then** e_2 **else** e_3	Conditional

For notational simplicity we assume that all functions and operations have the same number of arguments, mainly to avoid subscripted subscripts. A function with fewer than k parameters can be padded to one of k parameters by adding dummy parameters to the left of the parameter list and adding dummy arguments to every use of the function symbol. An actual implementation of an analysis based on this framework should not make this

assumption, nor should it assume that the functions are named $f_1, \ldots, f_n$, parameters $x_1, \ldots, x_k$, and basic operations $a_1, a_2, \ldots$.

The language is in a sense weakly typed since each function has a fixed arity, *i.e.* number of arguments. Result of functions can, on the other hand, be functions partially applied to various numbers of arguments.

Example As an example of how to use the curried style in the language, consider the following program.

$$\begin{array}{ll} f & = taut\ g\ 2 \\ g\ x\ y & = x \wedge \neg y \\ taut\ h\ n & = \textbf{if}\ n = 0\ \textbf{then}\ h \\ & \quad \textbf{else}\ taut\ (h\ true)\ (n-1) \wedge taut\ (h\ false)\ (n-1) \end{array}$$

The function *taut* is a function in two arguments. If h is an n-argument function then a call $taut\ h\ n$ returns $true$ if h is a tautology, otherwise it returns $false$.

2 Closure semantics

We have not specified which basic operations and datatypes the language should contain. The details are not important as long as the underlying datatype contains the truth values. We will therefore assume that the language has an underlying datatype represented with the set $Base$ and that the basic operations $a_1, a_2, \ldots$ have standard meanings as functions $\underline{a}_i : Base^k \to Base_\perp$. This is actually a restriction on the language as we do not allow basic operations on higher order objects. This means that we cannot have, say, `map` or `mapcar` as basic operations (but it will be easy to define those as user defined functions).

2.1 Closures

The semantics presented below is based on "closures" so a partially applied function will be represented as a closure or tuple $[i, v_1, \ldots, v_j]$ of the function identification i for the function name f_i and the so far computed arguments $(v_1, \ldots, v_j)$. Only when all arguments are provided, *i.e.* when $j = k$ will

the expression e_i defining the i^{th} function be evaluated. The arguments in the closure may belong to the base type but, as the language is higher-order, they may also be closures themselves. The set of values that may appear in computations may then be defined recursively as either base type values or closures of values:

$$V \quad = Base \cup (\{1, \ldots, n\} \times V^*)$$

We here use the numbers $1, \ldots, n$ instead of the function names so as to remove the syntactic information from the semantic domains. As an example $[i, \epsilon]$, with ϵ as the empty sequence, denotes the i^{th} function before being applied to any arguments. We will use square brackets to construct and denote closures in expressions like $[i, v_1, \ldots, v_\ell]$. In general we use subscripts to name (and distinguish) elements in tuples and a down arrow ($\downarrow$) to extract elements from tuples. Closures do not contain any information about the results from calling the (partially applied) functions. This means that there is no natural ordering between closures or base values and we may use $V_\perp$ as a flat domain.

2.2 Fixpoint semantics with closures

Semantic domains

$$\begin{array}{lll} V & = Base \cup (\{1, \ldots, n\} \times V^*) & v, w \in V_\perp, \quad \rho \in V^k \\ \Phi & = V^k \to V_\perp & \phi \in \Phi^n \end{array}$$

Semantic functions.

$$\begin{array}{lll} \mathbf{E}_c[\![e]\!] : \Phi^n \to V^k \to V_\perp & & \text{Expression meanings} \\ \mathbf{U}_c[\![p]\!] : \Phi^n & & \text{Program meanings} \\ \underline{a}_j \quad : V^k \to V_\perp & & \text{Basic operations} \end{array}$$

with definitions:

$$
\begin{aligned}
\mathsf{E}_c[\![x_i]\!]\phi\rho = \quad & \rho \downarrow i \\
\mathsf{E}_c[\![f_i]\!]\phi\rho = \quad & [i, \epsilon] \\
\mathsf{E}_c[\![a_j(e_1, \ldots, e_k)]\!]\phi\rho = \quad & \\
& \textbf{let } v_i = \mathsf{E}_c[\![e_i]\!]\phi\rho \textbf{ for } i = 1, \ldots, k \textbf{ in} \\
& \textbf{if } v_i \notin \mathit{Base} \textbf{ for some } i \textbf{ then } \bot \textbf{ else } \underline{a}_j(v_1, \ldots, v_k) \\
\mathsf{E}_c[\![e_1(e_2)]\!]\phi\rho = \quad & \textbf{let } w_1 = \mathsf{E}_c[\![e_1]\!]\phi\rho \textbf{ and } w_2 = \mathsf{E}_c[\![e_2]\!]\phi\rho \textbf{ in} \\
& \textbf{if } w_1 = \bot \textbf{ or } w_2 = \bot \textbf{ or } w_1 \in \mathit{Base} \textbf{ then } \bot \textbf{ else} \\
& \textbf{let } [i, v_1, \ldots, v_\ell] = w_1 \textbf{ in} \\
& \textbf{if } \ell + 1 < k \textbf{ then } [i, v_1, \ldots, v_\ell, w_2] \\
& \textbf{else } \phi_i(v_1, \ldots, v_\ell, w_2) \\
\mathsf{E}_c[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\phi\rho = \quad & \\
& \textbf{if } \mathsf{E}_c[\![e_1]\!]\phi\rho = \mathit{true} \textbf{ then } \mathsf{E}_c[\![e_2]\!]\phi\rho \textbf{ else} \\
& \textbf{if } \mathsf{E}_c[\![e_1]\!]\phi\rho = \mathit{false} \textbf{ then } \mathsf{E}_c[\![e_3]\!]\phi\rho \textbf{ else } \bot
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{U}_c[\![f_1\ x_1 \cdots x_k = e_1 \quad \ldots \quad f_n\ x_1 \cdots x_k = e_n]\!] = \\
&\qquad \textbf{fix } \lambda\phi.\langle \mathsf{E}_c[\![e_1]\!]\phi, \ldots, \mathsf{E}_c[\![e_n]\!]\phi \rangle
\end{aligned}
$$

2.3 Comment

In a higher-order semantics for this type of language, functions denote functions. The meaning of a partial application is thus a function of the remaining arguments to a result. The closure semantics will in a sense contain both more and less information. It contains more in that the functional values contain information about which of the user-defined functions the value originated from. It contains less since a closure in itself does not describe the functional value. Only a closure together with the function environment will contain this information.

3 Minimal function graph

We will now present the minimal function graph semantics for the language. The function environment is somewhat special in this semantics as the functions are not represented by functions but by sets of argument-result pairs.

These will be the smallest sets sufficient to include all calls resulting from applying the program to an *initial call description*. This is of the form $\langle c_1, \ldots, c_n \rangle$ where each c_i is the set of arguments with which f_i is called externally.

3.1 Function environment

As indicated earlier, in the minimal function graph approach a function is represented as a set of argument-result pairs. We will use the power-set

$$\Psi = \mathcal{P}(V^k \times V_\perp)$$

to represent the meaning of functions. In the minimal function graph approach the function environment plays a dual role. It keeps the results of function calls and, as well, holds the list of "needed" calls. The "argument needs" is the set of argument tuples on which a function must be evaluated to complete the computation. They may be described in a power-set

$$C = \mathcal{P}(V^k)$$

A connection between Ψ and C can be given with the following two functions.

$$\begin{array}{ll} \mathit{getcalls} : \Psi \to C & \mathit{getcalls}(\psi) = \{\vec{v} \mid \langle \vec{v}, r \rangle \in \psi\} \\ \mathit{savecalls} : C \to \Psi & \mathit{savecalls}(c) = \{\langle \vec{v}, \perp \rangle \mid \vec{v} \in c\} \end{array}$$

If C is ordered with set-inclusion and function denotations Ψ are ordered with the ordering

$$\begin{array}{l} \psi_1 \sqsubseteq \psi_2 \\ \Updownarrow \\ \forall \langle \vec{v}, r \rangle \in \psi_1 . \langle \vec{v}, r \rangle \in \psi_2 \vee (r = \perp \wedge \langle \vec{v}, s \rangle \in \psi_2 \text{ for some } s \in V) \end{array}$$

then C may be seen as an abstraction of the domain of function denotations Ψ. Clearly $\mathit{getcalls} \circ \mathit{savecalls}$ is the identity on C and $\forall \psi \in \Psi.\ \mathit{savecalls}(\mathit{getcalls}(\psi)) \sqsubseteq \psi$. The functions *getcalls* and *savecalls* are extended to Ψ^n and C^n component-wise.

3.2 Fixpoint semantics with minimal function graphs

We are now ready to introduce the minimal function graph semantics by defining two semantic functions $\mathbf{E}_m$ and $\mathbf{U}_m$. The function $\mathbf{E}_m$ has two uses: to evaluate; and to collect function arguments needed to do the evaluation.

Semantic domains

V	$= Base \cup (\{1,\ldots,n\} \times V^*)$	Closures	
C	$= \mathcal{P}(V^k)$	Sets of arguments	$\vec{c} \in C^n$
Ψ	$= \mathcal{P}(V^k \times V_\perp)$	Function graphs	$\psi \in \Psi^n$

Semantic functions

$\mathbf{E}_m[\![e]\!] : \Psi^n \to V^k \to V_\perp \times C^n$
$\mathbf{U}_m[\![p]\!] : C^n \to \Psi^n$

The function $\mathbf{U}_m$ maps program input needs to an n-tuple of minimal function graphs, one for each f_i.

$$\mathbf{E}_m[\![x_i]\!]\psi\rho = \quad \langle \rho \downarrow i, nocalls\rangle$$

$$\mathbf{E}_m[\![f_i]\!]\psi\rho = \quad \langle [i,\epsilon], nocalls\rangle$$

$$\begin{aligned}\mathbf{E}_m[\![a_j(e_1,\ldots,e_k)]\!]\psi\rho = \\ &\textbf{let } \langle w_i,\vec{c}_i\rangle = \mathbf{E}_m[\![e_i]\!]\psi\rho \textbf{ for } i=1,\ldots,k \textbf{ in}\\ &\textbf{if } w_i \notin Base \textbf{ for some } i \textbf{ then } \langle \perp, \vec{c}_1 \sqcup \cdots \sqcup \vec{c}_k\rangle\\ &\textbf{else } \langle \underline{a}_j(w_1,\ldots,w_k), \vec{c}_1 \sqcup \cdots \sqcup \vec{c}_k\rangle\end{aligned}$$

$$\begin{aligned}\mathbf{E}_m[\![e_1(e_2)]\!]\psi\rho = &\textbf{let } \langle w_1,\vec{c}_1\rangle = \mathbf{E}_m[\![e_1]\!]\psi\rho \textbf{ and}\\ &\quad \langle w_2,\vec{c}_2\rangle = \mathbf{E}_m[\![e_2]\!]\psi\rho \textbf{ in}\\ &\textbf{if } w_1 = \perp \textbf{ or } w_2 = \perp \textbf{ or } w_1 \in Base \textbf{ then } \langle \perp, \vec{c}_1 \sqcup \vec{c}_2\rangle\\ &\textbf{else let } [i,v_1,\ldots,v_\ell] = w_1 \textbf{ in}\\ &\textbf{if } \ell+1 < k \textbf{ then } \langle [i,v_1,\ldots,v_\ell,w_2], \vec{c}_1 \sqcup \vec{c}_2\rangle\\ &\textbf{else } \langle lookup_i(\langle v_1,\ldots,v_\ell,w_2\rangle,\psi),\\ &\qquad \vec{c}_1 \sqcup \vec{c}_2 \sqcup only_i(\langle v_1,\ldots,v_\ell,w_2\rangle)\rangle\end{aligned}$$

$$\begin{aligned}\mathbf{E}_m[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\psi\rho = \\ &\textbf{let } \langle w_1,\vec{c}_1\rangle = \mathbf{E}_m[\![e_1]\!]\psi\rho\\ &\quad \langle w_2,\vec{c}_2\rangle = \mathbf{E}_m[\![e_2]\!]\psi\rho\\ &\quad \langle w_3,\vec{c}_3\rangle = \mathbf{E}_m[\![e_3]\!]\psi\rho \textbf{ in}\\ &\textbf{if } w_1 = true \textbf{ then } \langle w_2, \vec{c}_1 \sqcup \vec{c}_2\rangle\\ &\textbf{else if } w_1 = false \textbf{ then } \langle w_3, \vec{c}_1 \sqcup \vec{c}_3\rangle \textbf{ else } \langle \perp, \vec{c}_1\rangle\end{aligned}$$

$$\begin{aligned}
&\mathbf{U}_m[\![f_1\ x_1 \cdots x_k = e_1 \quad \ldots \quad f_n\ x_1 \cdots x_k = e_n]\!]\vec{c} = \\
&\qquad \mathbf{fix}\lambda\psi.\, savecalls(\vec{c}) \sqcup \\
&\qquad\qquad \langle map_1(\mathbf{E}_m[\![e_1]\!]\psi, getcalls(\psi) \downarrow 1), \ldots, \\
&\qquad\qquad\ map_1(\mathbf{E}_m[\![e_n]\!]\psi, getcalls(\psi) \downarrow n)\rangle \sqcup \\
&\qquad\qquad savecalls(map_2(\mathbf{E}_m[\![e_1]\!]\psi, getcalls(\psi) \downarrow 1)) \sqcup \cdots \sqcup \\
&\qquad\qquad savecalls(map_2(\mathbf{E}_m[\![e_n]\!]\psi, getcalls(\psi) \downarrow n))
\end{aligned}$$

with $lookup : V^k \times \Psi \to V_\perp$ and $only_i : V^k \to C^n$

$$\begin{aligned}
&lookup_i(\vec{v}, \psi) = \bigsqcup\{r \mid \langle \vec{v}, r\rangle \in (\psi \downarrow i)\} \\
&only_i(\vec{v}) = \langle \underbrace{\emptyset, \ldots, \emptyset}_{i-1}, \{\vec{v}\}, \emptyset, \ldots, \emptyset\rangle \\
&nocalls = \langle \underbrace{\emptyset, \ldots, \emptyset}_{n}\rangle \\
&map_1(f, s) = \{\langle \vec{v}, r\rangle \mid \vec{v} \in s \wedge f(\vec{v}) = \langle r, _\rangle\} \\
&map_2(f, s) = \bigsqcup\{c \mid \vec{v} \in s \wedge f(\vec{v}) = \langle _, c\rangle\}
\end{aligned}$$

In words the fixpoint computation says that the function environment should contain: all the original calls ($savecalls(\vec{c})$), results from functions with the arguments in the environment ($map_1(\mathbf{E}_m[\![e_i]\!]\psi, getcalls(\psi) \downarrow i)$), and all new calls found when calling the functions with the arguments in the environment ($savecalls(map_2(\mathbf{E}_m[\![e_i]\!]\psi, getcalls(\psi) \downarrow i))$).

3.3 Relationship

The two semantics may be proven equivalent in the sense that all function results that may be computed by either semantics may also be computed by the other. This may be expressed as

$$\forall p, \vec{v}, j.(\mathbf{U}_c[\![p]\!] \downarrow j)\vec{v} = lookup_j(\vec{v}, \mathbf{U}_m[\![p]\!]only_j(\vec{v}))$$

This may be done as for the first-order case in [5] where also the stronger condition that the minimal function graph semantics only contains the needed computations is proved.

3.4 Comment

We have esentially used the same ordering on the domain Ψ as one would have on the function domain $V^k \to V_\perp$ and it could have been tempting to

use a function domain rather than a power-set constuction with an unusual ordering. It is however important to distinguaish between a function being called and returning undefined and a function not being called at all. This is described in our power-set construction by the absence or presence of an argument tuple in the set but it is not registered in the domain $V^k \to V_\perp$. A different description would be to use a function domain with a doubble lifted result domain $V^k \to V_{!\perp}$ where the lower bottom ($\perp$) indicates that the function has not been called for the given argument tuple (absence in the graph) and the second bottom value (!) indicates that the function is undefined for the given argument tuple. The latter approach was taken in Jones and Mycroft's paper on minimal function graphs for a first-order language [4] whereas our approach is closer to Cousots' description of chaotic fixpoint iteration [3].

4 Closure analysis

As an example of how one may use this semantics in program analysis we will here sketch a closure analysis for the language. A closure analysis will for each use of a functional expression compute a superset (usually small) of the user defined functions that the expression may yield as value. The result of a closure analysis provides control-flow information which may greatly simplify later data-flow analyses of such higher-order languages. A closure analysis for a strict language was first constructed by Peter Sestoft [6]. He later extended the work to a lazy language in his PhD thesis. Closure analysis is central to the working of the Similix partial evaluator [1], and was independently rediscovered by Shivers [7].

4.1 Abstract closures

An abstract closure is a pair of a function identification in $\{1, \ldots, n\}$ and a number of provided arguments $\{0, \ldots, k-1\}$. An abstract value is either a set of abstract closures or the symbol $\underline{atom}$ denoting that the value is of base type. The domain of abstract values will be denoted $\tilde{V}$.

$$\tilde{V} = \mathcal{P}(\{\underline{atom}\} \cup (\{1, \ldots, n\} \times \{0, \ldots, k-1\}))$$

An abstract closure will then no longer contain information about values of arguments to partially applied functions. This information may, however, be

retrieved from a function environment so an abstract closure $[i, j]$ denotes the set of all closures built from the i^{th} function with j arguments whose descriptions may in turn be found in the function environment. More formally we define the domain of abstract function environments to be

$$\tilde{\Psi} = (\tilde{V}^k \times \tilde{V})$$

4.2 Relating abstract and concrete closures

Abstract closures only contain information about the number of arguments to a function. The values of arguments is described in the abstract function environment or recorded as abstract needs. Due to this it is natural to define two abstraction functions with functionalities

$$\alpha_1 : V\_\bot \to \tilde{V}$$
$$\alpha_2 : V_\bot \to (\tilde{V}^k)^n$$

and definitions

$$\alpha_1([i, v_1, \ldots, v_j]) = \{[i, j]\}$$
$$\alpha_1(c) = \{\underline{atom}\} \qquad c \in Base$$
$$\alpha_1(\bot) = \emptyset$$
$$\alpha_2([i, v_1, \ldots, v_j]) =$$
$$\langle \underbrace{\langle \emptyset, \ldots, \emptyset \rangle, \ldots}_{i-1}, \langle \alpha_1(v_1), \ldots, \alpha_1(v_j), \emptyset, \ldots \rangle, \langle \emptyset, \ldots, \emptyset \rangle, \ldots \rangle$$
$$\alpha_2(c) = \langle \langle \emptyset, \ldots, \emptyset \rangle, \ldots \rangle \qquad c \in Base$$
$$\alpha_2(\bot) = \langle \langle \emptyset, \ldots, \emptyset \rangle, \ldots \rangle$$

The abstraction functions α_1 and α_2 may be extended to $\mathcal{P}(V)$ as the $\cup$ and $\sqcup$-closure, respectively so that we have the functionalities

$$\alpha_1 : \mathcal{P}(V) \to \tilde{V}$$
$$\alpha_2 : \mathcal{P}(V) \to (\tilde{V}^k)^n$$

4.3 Closure analysis

The closure analysis should map a program into an abstract function environment which safely approximates the minimal function graph interpretation.

Semantic domains.

$\tilde{V}$	Abstract closures	$\tilde{v}, \tilde{w} \in \tilde{V}_\perp, \tilde{\rho} \in \tilde{V}^k$
$\tilde{\Psi}^n$	Function environment	$\tilde{\psi} \in \tilde{\Psi}^n$
$(\tilde{V}^k)^n$	Argument needs	$\tilde{c} \in (\tilde{V}^k)^n$

Semantic functions.

$\tilde{\mathbf{E}}[\![e]\!] : \tilde{\Psi}^n \to \tilde{V}^k \to \tilde{V} \times (\tilde{V}^k)^n$

$\tilde{\mathbf{U}}[\![p]\!] : (\tilde{V}^k)^n \to \tilde{\Psi}^n$

Definitions.

$$\tilde{\mathbf{E}}[\![x_i]\!]\tilde{\psi}\tilde{\rho} = \quad \langle \tilde{\rho} \downarrow i, nocalls \rangle$$

$$\tilde{\mathbf{E}}[\![f_i]\!]\tilde{\psi}\tilde{\rho} = \quad \langle \{[i,0]\}, nocalls \rangle$$

$$\begin{aligned}\tilde{\mathbf{E}}[\![a_j(e_1,\ldots,e_k)]\!]\tilde{\psi}\tilde{\rho} = {} & \textbf{let } \langle \tilde{w}_i, \tilde{c}_i \rangle = \tilde{\mathbf{E}}[\![e_i]\!]\tilde{\psi}\tilde{\rho} \textbf{ for } i = 1,\ldots,k \textbf{ in} \\ & \textbf{if } \underline{atom} \notin \tilde{w}_i \textbf{ for some } i \textbf{ then } \langle \emptyset, \tilde{c}_1 \sqcup \cdots \sqcup \tilde{c}_k \rangle \\ & \textbf{else } (\{\underline{atom}\}, \tilde{c}_1 \sqcup \cdots \sqcup \tilde{c}_k)\end{aligned}$$

$$\begin{aligned}\tilde{\mathbf{E}}[\![e_1(e_2)]\!]\tilde{\psi}\tilde{\rho} = {} & \textbf{let } \langle \tilde{w}_1, \tilde{c}_1 \rangle = \tilde{\mathbf{E}}[\![e_1]\!]\tilde{\psi}\tilde{\rho} \textbf{ and} \\ & \quad \langle \tilde{w}_2, \tilde{c}_2 \rangle = \tilde{\mathbf{E}}[\![e_2]\!]\tilde{\psi}\tilde{\rho} \textbf{ in} \\ & \textbf{if } \tilde{w}_2 = \emptyset \textbf{ then } \langle \perp, \tilde{c}_1 \sqcup \tilde{c}_2 \rangle \textbf{ else} \\ & \langle \textstyle\bigcup_{[i,j]\in\tilde{w}_1} \{ \textbf{if } j = k-1 \textbf{ then } \widetilde{lookup}_i(\tilde{\psi}) \\ & \qquad \textbf{else } [i, j+1]\}, \\ & \tilde{c}_1 \sqcup \tilde{c}_2 \sqcup \textstyle\bigsqcup_{[i,j]\in\tilde{w}_1} \widetilde{only}_i(\langle \underbrace{\emptyset,\ldots,\emptyset}_{j}, \tilde{w}_2, \emptyset, \ldots \rangle) \rangle\end{aligned}$$

$$\begin{aligned}\tilde{\mathbf{E}}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\tilde{\psi}\tilde{\rho} = {} & \\ & \textbf{let } \langle \tilde{w}_1, \tilde{c}_1 \rangle = \tilde{\mathbf{E}}[\![e_1]\!]\tilde{\psi}\tilde{\rho} \\ & \quad \langle \tilde{w}_2, \tilde{c}_2 \rangle = \tilde{\mathbf{E}}[\![e_2]\!]\tilde{\psi}\tilde{\rho} \\ & \quad \langle \tilde{w}_3, \tilde{c}_3 \rangle = \tilde{\mathbf{E}}[\![e_3]\!]\tilde{\psi}\tilde{\rho} \textbf{ in} \\ & \textbf{if } \underline{atom} \in \tilde{w}_1 \textbf{ then } \langle \tilde{w}_2 \cup \tilde{w}_3, \tilde{c}_1 \sqcup \tilde{c}_2 \sqcup \tilde{c}_3 \rangle \\ & \textbf{else } (\emptyset, \tilde{c}_1)\end{aligned}$$

$$\begin{aligned}\tilde{\mathbf{U}}[\![f_1\ x_1 \cdots x_k = e_1 \quad \ldots \quad f_n\ x_1 \cdots x_k = e_n]\!]\tilde{c} = {} & \\ & \textbf{fix}\lambda\tilde{\psi}.\, \widetilde{savecalls}(\tilde{c}) \sqcup \\ & \quad \langle \widetilde{map}_1(\tilde{\mathbf{E}}[\![e_1]\!]\tilde{\psi}, \widetilde{getcalls}(\tilde{\psi}) \downarrow 1), \ldots, \\ & \quad \widetilde{map}_1(\tilde{\mathbf{E}}[\![e_n]\!]\tilde{\psi}, \widetilde{getcalls}(\tilde{\psi}) \downarrow n) \rangle \sqcup \\ & \quad \widetilde{savecalls}(\widetilde{map}_2(\tilde{\mathbf{E}}[\![e_1]\!]\tilde{\psi}, \widetilde{getcalls}(\tilde{\psi}) \downarrow 1)) \sqcup \cdots \sqcup \\ & \quad \widetilde{savecalls}(\widetilde{map}_2(\tilde{\mathbf{E}}[\![e_n]\!]\tilde{\psi}, \widetilde{getcalls}(\tilde{\psi}) \downarrow n))\end{aligned}$$

with

$$\widetilde{lookup}_i(\tilde{\psi}) = \textbf{let } \langle\langle \tilde{v}_1, \ldots, \tilde{v}_k\rangle, \tilde{v}_r\rangle = \tilde{\psi} \downarrow i \textbf{ in } \tilde{v}_r$$
$$\widetilde{only}_i(\tilde{v}) = \langle \underbrace{noargs, \ldots, noargs}_{i-1}, \tilde{v}, noargs, \ldots\rangle$$
$$noargs = \langle \underbrace{\emptyset, \ldots, \emptyset}_{k}\rangle$$
$$nocalls = \langle \underbrace{noargs, \ldots, noargs}_{n}\rangle$$
$$\widetilde{map}_1(f, \tilde{v}) = \langle \tilde{v}, f(\tilde{v}) \downarrow 1\rangle$$
$$\widetilde{map}_2(f, \tilde{v}) = f(\tilde{v}) \downarrow 2$$
$$\widetilde{savecalls}(\langle \tilde{v}_1, \ldots, \tilde{v}_n\rangle) = \langle\langle \tilde{v}_1, \emptyset\rangle, \ldots, \langle \tilde{v}_n, \emptyset\rangle\rangle$$
$$\widetilde{getcalls}(\langle\langle \tilde{v}_1, \tilde{r}_1\rangle, \ldots, \langle \tilde{v}_n, \tilde{r}_n\rangle\rangle) = \langle \tilde{v}_1, \ldots, \tilde{v}_n\rangle$$

4.4 Example

Consider the tautology function from the introduction

$$\begin{array}{ll} f & = taut\ g\ 2 \\ g\ x\ y & = x \wedge \neg y \\ taut\ h\ n & = \textbf{if } n = 0 \textbf{ then } h \textbf{ else} \\ & \quad taut\ (h\ true)\ (n-1) \wedge taut\ (h\ false)\ (n-1) \end{array}$$

The initial call description states that only the function f may be called externally. With this the closure analysis computes the following function environment:

$$\begin{array}{ll} f: & \langle\{\underline{atom}\}\rangle \\ g: & \langle\{\underline{atom}\}, \{\underline{atom}\}, \{\underline{atom}\}\rangle \\ taut: & \langle\{[g,0], [g,1], \underline{atom}\}, \{\underline{atom}\}, \{\underline{atom}\}\rangle \end{array}$$

4.5 Safety

The safety condition states that the closure analysis from a call description will compute a superset of the set of closures that an expression may yield as value. The proof will be based on fixpoint induction where a relation between the semantic functions $\mathbf{E}_m$ and $\tilde{\mathbf{E}}$ is lifted to a relation between $\mathbf{U}_m$ and $\tilde{\mathbf{U}}$.

Abstraction functions We have already introduced the abstraction function for closures and sets of closures but we also need the extension to argument needs and function environments.

$$
\begin{array}{l}
\alpha_c : C^n \to (\tilde{V}^k)^n \\
\alpha_c(\langle c_1, \ldots, c_n \rangle) = \langle \langle \alpha_1(\{\rho \downarrow 1 \mid \rho \in c_1\}), \ldots, \alpha_1(\{\rho \downarrow k \mid \rho \in c_1\}) \rangle, \ldots \rangle \\
\alpha_\psi : \Psi^n \to \tilde{\Psi}^n \\
\alpha_\psi(\psi) = \widetilde{savecalls}(\alpha_c(getcalls(\psi))) \sqcup \\
\qquad\qquad \widetilde{savecalls}(\bigsqcup_i (\alpha_2(\{r \mid \langle \_, r \rangle \in \psi \downarrow i\} \setminus \{\bot\}))) \\
\qquad\qquad \langle \langle \emptyset, \ldots, \emptyset, \alpha_1(\{r \mid \langle \_, r \rangle \in \psi \downarrow 1\}) \rangle, \ldots \rangle
\end{array}
$$

The abstraction of a function environment will, in other words, abstract calls recorded as arguments, calls recorded in function results and abstractions of function results.

Local safety The relationship between $\mathbf{E}_m$ and $\tilde{\mathbf{E}}$ states that for related function and parameter environments the functions should produce related results. Values are related if the abstraction of values from the MFG-semantics is less than or equal to values produced by the closure semantics.

Given function and parameter environments

$$\rho \in V^k, \quad \tilde{\rho} \in \tilde{V}^k, \quad \psi \in \Psi^n, \quad \tilde{\psi} \in \tilde{\Psi}^n$$

where

$$
\begin{array}{ll}
\alpha_1(\rho \downarrow i) \subseteq \tilde{\rho} \downarrow i & \forall i \in \{1, \ldots, k\} \\
(\bigsqcup_i \alpha_2(\rho \downarrow i)) \sqsubseteq \widetilde{getcalls}(\tilde{\psi}) & \forall i \in \{1, \ldots, k\} \\
\alpha_\psi(\psi) \sqsubseteq \tilde{\psi} &
\end{array}
$$

the safety condition for the semantic function $\tilde{\mathbf{E}}$ is

$$
\begin{array}{l}
\alpha_1(\{\mathbf{E}_m[\![e]\!]\psi\rho \downarrow 1\} \setminus \{\bot\}) \subseteq \tilde{\mathbf{E}}[\![e]\!]\tilde{\psi}\tilde{\rho} \downarrow 1 \quad \wedge \\
\alpha_c(\mathbf{E}_m[\![e]\!]\psi\rho \downarrow 2) \sqsubseteq \tilde{\mathbf{E}}[\![e]\!]\tilde{\psi}\tilde{\rho} \downarrow 2 \sqcup \widetilde{getcalls}(\tilde{\psi})
\end{array}
$$

for any expression e.

The second requirement on environments states that all partial function applications recorded in the parameter environment must also be recorded in the abstract function environment. The second safety condition states that needs will either be computed by the $\tilde{\mathbf{E}}$ function or be present in the abstract function environment.

Proof The proof is by structural induction over expressions. The proof for the first component of the functions is fairly straight forward. The induction start involves $\alpha_1(\rho{\downarrow}i) \subseteq \tilde{\rho}{\downarrow}i$ and $\alpha_1([i,\epsilon]) \subseteq \{[i,0]\}$. In the induction step for standard operations we have $\alpha_1(\bot) \subseteq \emptyset$ and $\alpha_1(\underline{a}_j(w_1,\ldots,w_k)) \subseteq \{\underline{atom}\}$. For function application let $w_1 = [i, v_1, \ldots, v_j]$ where $\alpha_1(w_1) = [i,j]$ it then follows that

$$\begin{array}{ll} \alpha_1([i,v_1,\ldots,v_j,w_2]) = \{[i,j+1]\} & if\ j < k-1 \\ \alpha_1(lookup_i(\langle v_1,\ldots,v_j,w_2\rangle,\psi) \subseteq \tilde{lookup}_i(\tilde{\psi}) & if\ j = k-1 \end{array}$$

For the second component the proof is complicated by the fact that argument needs are recorded earlier in the closure analysis than in the MFG-semantics where only full applications result in an argument tuple being recorded as needed. The result may, however, again be proved by structural induction over expressions. For the induction start we have that $\alpha_c(nocalls) = nocalls$

For function applications it involves proving the relation

$$\begin{array}{l} \alpha_c(only_i(\langle v_1,\ldots,v_\ell,w_2\rangle)) \sqsubseteq \\ \quad \widetilde{only}_i(\langle\emptyset,\ldots,\emptyset,\tilde{w}_2,\emptyset,\ldots\rangle) \sqcup c_1 \sqcup c_2 \sqcup \widetilde{getcalls}(\tilde{\psi}) \end{array}$$

where the closure $[i, v_1, \ldots, v_\ell]$ is the MFG-result of a subexpression with abstract needs c_1. A closure may be produced in four different ways: as the value of a parameter, as the result of a full function application, as a function symbol and as a partial application. In the first two cases the assumed relationship between environments guarrantee the relations. In the third case there is nothing to prove since ℓ is zero and in the fourth case it follows from the structural induction over expressions.

Global safety The global safety condition states that the semantic function $\tilde{\mathbf{U}}$ from a safe description of initial calls will compute a safe description of argument and result closures for the functions.

$$\forall \widetilde{c} \in (\tilde{V}^k)^n, c \in C^n.\ \alpha_c(c) \sqsubseteq \widetilde{c} \Rightarrow \alpha_\psi(\mathbf{U}_m[\![p]\!]c) \sqsubseteq \tilde{\mathbf{U}}[\![p]\!]\widetilde{c}$$

Proof Define the functions

$$\begin{aligned}
F_c(\psi) = {} & savecalls(\vec{c}) \sqcup \\
& \langle map_1(\mathsf{E}_m[\![e_1]\!]\psi, getcalls(\psi) \downarrow 1), \ldots, \\
& \ map_1(\mathsf{E}_m[\![e_n]\!]\psi, getcalls(\psi) \downarrow n)\rangle \sqcup \\
& savecalls(map_2(\mathsf{E}_m[\![e_1]\!]\psi, getcalls(\psi) \downarrow 1)) \sqcup \cdots \sqcup \\
& savecalls(map_2(\mathsf{E}_m[\![e_n]\!]\psi, getcalls(\psi) \downarrow n)) \\
\tilde{F}_{\tilde{c}}(\tilde{\psi}) = {} & \widetilde{savecalls}(\tilde{c}) \sqcup \\
& \langle \widetilde{map}_1(\tilde{\mathsf{E}}[\![e_1]\!]\tilde{\psi}, \widetilde{getcalls}(\tilde{\psi}) \downarrow 1), \ldots, \\
& \ \widetilde{map}_1(\tilde{\mathsf{E}}[\![e_n]\!]\tilde{\psi}, \widetilde{getcalls}(\tilde{\psi}) \downarrow n)\rangle \sqcup \\
& \widetilde{savecalls}(\widetilde{map}_2(\tilde{\mathsf{E}}[\![e_1]\!]\tilde{\psi}, \widetilde{getcalls}(\tilde{\psi}) \downarrow 1)) \sqcup \cdots \sqcup \\
& \widetilde{savecalls}(\widetilde{map}_2(\tilde{\mathsf{E}}[\![e_n]\!]\tilde{\psi}, \widetilde{getcalls}(\tilde{\psi}) \downarrow n))
\end{aligned}$$

The proof will be based on fixpoint induction since

$$\begin{aligned}
&\mathsf{U}_m[\![p]\!]c = fix(F_c) \\
&\tilde{\mathsf{U}}[\![p]\!]\tilde{c} = fix(\tilde{F}_{\tilde{c}})
\end{aligned}$$

Let c and $\tilde{c}$ be given and assume that they satisfy $\alpha_c(c) \sqsubseteq \tilde{c}$. For the induction step we will assume that ψ and $\tilde{\psi}$ are given and that they satisfy $\alpha_\psi(\psi) \sqsubseteq \tilde{\psi}$. We then want to prove that $\alpha_\psi(F_c(\psi)) \sqsubseteq \tilde{F}_{\tilde{c}}(\tilde{\psi})$.

The proof of the induction step will follow the components of the functions F and $\tilde{F}$. The first part is to see that

$$\alpha_c(getcalls(savecalls(c)) \sqsubseteq \widetilde{getcalls}(\widetilde{savecalls}(\tilde{c}))$$

which is satisfied since $getcalls \circ savecalls$ is the identity function.

For the two remaining parts of the function F let $i \in \{1, \ldots, n\}$ and $\rho \in getcalls(\psi) \downarrow i$. Now let $\tilde{\rho} = \widetilde{getcalls} \downarrow i$. From the assumption we then know that $\forall j.\alpha_1(\rho \downarrow j) \sqsubseteq \tilde{\rho} \downarrow j$. Furthermore $\forall j.\alpha_2(\rho \downarrow j) \sqsubseteq \alpha_c(getcalls(\psi)) \sqsubseteq \widetilde{getcalls}(\tilde{\psi})$. We may therefore use the local safety condition to establish the induction step.

5 Conclusion

The paper describes a minimal function graph semantics for a higher order strict functional language. The minimal function graph semantics may be

used as a basis for program analysis. This is illustrated by the construction of a closure analysis for the language and the proof of safety of the closure analysis with respect to the minimal function graph semantics.

Acknowledgement. The work was supported in part by the Danish Research Council under the DART project.

References

[1] Anders Bondorf. Similix user manual. Tech. rep., DIKU, Univ. of Copenhagen, Denmark, 1993.

[2] G L Burn, C L Hankin, and S Abramsky. Strictness analysis for higher-order functions. *Sci. Comp. Prog.*, 7:249–278, 1986.

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

[4] Neil D Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *13th POPL, St. Petersburg, Florida*, pages 296–306, January 1986.

[5] Alan Mycroft and Mads Rosendahl. Minimal function graphs are not instrumented. In *WSA'92, Bordeaux, France*, Bigre, pages 60–67. Irisa, Rennes, France, September 1992.

[6] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, Univ. of Copenhagen, Denmark, October 1988.

[7] O Shivers. Control flow analysis in scheme. In *SIGPLAN '88 Conference on PLDI, Atlanta, Georgia*, volume 23(7) of *ACM SIGPLAN Not.*, pages 164–174, July 1988.