Higher-Order Minimal Function Graphs

Neil D Jones and Mads Rosendahl

Datalogisk Institut, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø
Denmark
{neil, rose}@diku.dk

Abstract. We present a minimal function graph semantics for a higher-order functional language with applicative evaluation order. 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

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

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:

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

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 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.

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

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 $[i, e_1, \ldots, e_\ell]$. 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. In the semantics we will use a function *isbase* to test whether a value in V is of base type or is a closure.

2.2 Fixpoint semantics with closures

Semantic domains

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

Semantic functions.

$$\begin{array}{ll} \mathbf{E}_c[\![e]\!] : \Phi^n \to V^k \to V_\perp & \text{Expression meanings} \\ \mathbf{U}_c[\![p]\!] : \Phi^n & \text{Program meanings} \end{array}$$

with definitions:

$$\begin{array}{l} \mathbf{E}_c[\![x_i]\!]\phi\rho = \quad \rho_i \\ \mathbf{E}_c[\![f_i]\!]\phi\rho = \quad [i, \epsilon] \\ \mathbf{E}_c[\![a_j(e_1,\ldots,e_k)]\!]\phi\rho = \\ \qquad \textbf{let } v_i = \mathbf{E}_c[\![e_i]\!]\phi\rho \textbf{ for } i = 1,\ldots,k \textbf{ in} \\ \qquad \textbf{if some } v_i = \perp \textbf{ then } \perp \textbf{ else } \underline{a}_j(v_1,\ldots,v_k) \\ \mathbf{E}_c[\![e_1(e_2)]\!]\phi\rho = \textbf{let } d_1 = \mathbf{E}_c[\![e_1]\!]\phi\rho \textbf{ and } d_2 = \mathbf{E}_c[\![e_2]\!]\phi\rho \textbf{ in} \\ \qquad \textbf{if } d_1 = \perp \textbf{ or } d_2 = \perp \textbf{ or } isbase(d_1) \textbf{ then } \perp \textbf{ else} \\ \qquad \textbf{let } [i, v_1, \ldots, v_\ell] = d_1 \textbf{ in if } \ell + 1 < k \textbf{ then } [i, v_1, \ldots, v_\ell, d_2] \\ \qquad \textbf{else } \phi_i(v_1, \ldots, v_\ell, d_2) \\ \mathbf{E}_c[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\phi\rho = \\ \qquad \textbf{if } \mathbf{E}_c[\![e_1]\!]\phi\rho = true \textbf{ then } \mathbf{E}_c[\![e_2]\!]\phi\rho \\ \qquad \textbf{if } \mathbf{E}_c[\![e_1]\!]\phi\rho = false \textbf{ then } \mathbf{E}_c[\![e_3]\!]\phi\rho \textbf{ else } \perp \end{array}$$

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

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 on the form $\langle c_1, \ldots, c_k \rangle$ 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

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

with the ordering

$$\psi_1 \sqsubseteq \psi_2 \Leftrightarrow \forall \langle \vec{v}, r \rangle \in \psi_1 . \langle \vec{v}, r \rangle \in \psi_2 \vee (r = \perp \wedge \langle \vec{v}, s \rangle \in \psi_2 \text{ for some } s \in V$$

to represent the meaning of functions. The domain Ψ^n will be used for function environments. The set Ψ may be used as a domain with set inclusion as partial ordering. 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

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

The needs for the tuple of program functions are C^n, a domain partially ordered by component-wise set inclusion. The domain C may be seen as an abstraction of the domain of function denotations Ψ using the following two functions.

$$\begin{array}{ll} \mathit{getcalls} : \Psi \to C & \mathit{getcalls}(\psi) = \{\vec{v} \mid \langle \vec{v}, r\rangle \in \psi_1\} \\ \mathit{savecalls} : C \to \Psi & \mathit{savecalls}(c) = \{\langle \vec{v}, \bot\rangle \mid \vec{v} \in c_1\} \end{array}$$

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 *getcall* and *setcall* 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 E_m and U_m. The function E_m has two uses: to evaluate; and to collect function arguments needed to do the evaluation.

Semantic domains

V	$= \mathit{Base} \cup (\{1, \ldots, n\} \times V^*)$	Closures
D	$= V_\bot \times C^n$	A result and a "needed calls" tuple
C	$= \mathcal{P}(V^k)$	Sets of arguments
Ψ	$= \mathcal{P}(V^k \times V_\bot)$	Function graphs

Semantic functions

$$\mathsf{E}_m[\![e]\!] : \Psi^n \to V^k \to D$$
$$\mathsf{U}_m[\![p]\!] : C^n \to \Psi^n$$

The function U_m maps program input needs to an n-tuple of minimal function graphs, one for each f_i.

$$
\begin{aligned}
&\mathbf{E}_m[\![x_i]\!]\psi\rho = && \langle \rho_i, \emptyset^n \rangle \\
&\mathbf{E}_m[\![f_i]\!]\psi\rho = && \langle [i, \epsilon], \emptyset^n \rangle \\
&\mathbf{E}_m[\![a_j(e_1, \ldots, e_k)]\!]\psi\rho = \\
&\qquad && \textbf{let } \langle d_i, \vec{c}_i \rangle = \mathbf{E}_m[\![e_i]\!]\psi\rho \textbf{ for } i = 1, \ldots, k \textbf{ in} \\
&\qquad && \textbf{if some } d_i = \bot \textbf{ then } \langle \bot, \vec{c}_1 \sqcup \cdots \sqcup \vec{c}_k \rangle \\
&\qquad && \textbf{else } \langle \underline{a}_j(d_1, \ldots, d_k), \vec{c}_1 \sqcup \cdots \sqcup \vec{c}_k \rangle \\
&\mathbf{E}_m[\![e_1(e_2)]\!]\psi\rho = && \textbf{let } \langle d_1, \vec{c}_1 \rangle = \mathbf{E}_m[\![e_1]\!]\psi\rho \textbf{ and} \\
&\qquad && \quad \langle d_2, \vec{c}_2 \rangle = \mathbf{E}_m[\![e_2]\!]\psi\rho \textbf{ in} \\
&\qquad && \textbf{if } d_1 = \bot \textbf{ or } d_2 = \bot \textbf{ or } isbase(d_1) \textbf{ then } \langle \bot, \vec{c}_1 \sqcup \vec{c}_2 \rangle \\
&\qquad && \textbf{else let } [i, v_1, \ldots, v_\ell] = d_1 \textbf{ in} \\
&\qquad && \textbf{if } \ell + 1 < k \textbf{ then } \langle [i, v_1, \ldots, v_\ell, d_2], \vec{c}_1 \sqcup \vec{c}_2 \rangle \\
&\qquad && \textbf{else } \langle \mathit{lookup}_i(\langle v_1, \ldots, v_\ell, d_2 \rangle, \psi), \\
&\qquad && \qquad \vec{c}_1 \sqcup \vec{c}_2 \sqcup \mathit{only}_i(\langle v_1, \ldots, v_\ell, d_2 \rangle) \rangle \\
&\mathbf{E}_m[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\psi\rho = \\
&\qquad && \textbf{let } \langle d_1, \vec{c}_1 \rangle = \mathbf{E}_m[\![e_1]\!]\psi\rho \\
&\qquad && \quad \langle d_2, \vec{c}_2 \rangle = \mathbf{E}_m[\![e_2]\!]\psi\rho \\
&\qquad && \quad \langle d_3, \vec{c}_3 \rangle = \mathbf{E}_m[\![e_3]\!]\psi\rho \textbf{ in} \\
&\qquad && \textbf{if } d_1 = \mathit{true} \textbf{ then } \langle d_2, \vec{c}_1 \sqcup \vec{c}_2 \rangle \\
&\qquad && \textbf{else if } d_1 = \mathit{false} \textbf{ then } \langle d_3, \vec{c}_1 \sqcup \vec{c}_3 \rangle \textbf{ else } \langle \bot, \vec{c}_1 \rangle
\end{aligned}
$$

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

with $\mathit{lookup} : V^k \times \Psi \to V_\perp$ and $\mathit{only}_i : V^k \to C^n$

$$
\begin{aligned}
&\mathit{lookup}_i(\vec{v}, \psi) = \bigsqcup\{r \mid \langle \vec{v}, r \rangle \in \psi_i\} \\
&\mathit{only}_i(\vec{v}) = \langle \underbrace{\emptyset, \ldots, \emptyset}_{i-1}, \{\vec{v}\}, \emptyset, \ldots, \emptyset \rangle \\
&\mathit{map}_1(f, s) = \{\langle \vec{v}, r \rangle \mid \vec{v} \in s \wedge f(\vec{v}) = \langle r, \_ \rangle\} \\
&\mathit{map}_2(f, s) = \bigsqcup\{c \mid \vec{v} \in s \wedge f(\vec{v}) = \langle \_, c \rangle\}
\end{aligned}
$$

In words the fixpoint computation says that the function environment should contain: all the original calls, results from functions with the arguments in the environment, and all new calls found when calling the functions with the arguments in the environment.

3.3 Safety

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

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

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. The proofs are included in an extended version of this paper.

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

The concretisation function is of functionality

$$\gamma : \tilde{\Psi} \times \tilde{V} \to \mathcal{P}(V)$$

and may be defined as the least function satisfying:

$$\gamma(\psi', S) = \bigcup_{[i,j] \in S} \{[i, v_1, \ldots, v_j] \mid v_1 \in \gamma(\psi', \psi'_i \downarrow 1), \ldots, v_j \in \gamma(\psi', \psi'_i \downarrow j)\}$$

for $\psi' \in \tilde{\Psi}^n$.

As the concretisation function depends on two arguments there will be two abstraction functions: a function generating abstract function environments and a function returning abstract closures.

$$\alpha_1([i, v_1, \ldots, v_j]) =$$
$$\langle \underbrace{\langle\langle\emptyset, \ldots, \emptyset\rangle, \emptyset\rangle, \ldots}_{i-1}, \langle\langle\alpha_2(v_1), \ldots, \alpha_2(v_j), \emptyset, \ldots\rangle, \emptyset\rangle, \langle\langle\emptyset, \ldots, \emptyset\rangle, \emptyset\rangle, \ldots\rangle$$
$$\sqcup\, \alpha_1(v_1) \sqcup \cdots \sqcup \alpha_1(v_j)$$
$$\alpha_2([i, v_1, \ldots, v_j]) = \{[i, j]\}$$

The abstraction and concretisation functions may be extended to base values by the definitions $\gamma(\psi', \underline{atom}) = Base$ for any ψ' and $\alpha_2(c) = \underline{atom}$ for $c \in Base$. The functions α_1 and α_2 may further be extended to $\mathcal{P}(V)$ as the $\sqcup$ and $\cup$-closure, respectively so that we have the functionalities

$$\begin{aligned}\gamma &: \tilde{\Psi} \times \tilde{V} \to \mathcal{P}(V)\\ \alpha_1 &: \mathcal{P}(V) \to \tilde{\Psi}\\ \alpha_2 &: \mathcal{P}(V) \to \tilde{V}\end{aligned}$$

which constitutes a Galois connection between $\tilde{\Psi} \times \tilde{V}$ and $\mathcal{P}(V)$.

$$\begin{aligned}&\alpha_1(\gamma(\psi', v')) \sqsubseteq \psi' && \psi' \in \tilde{\Psi}, \quad v' \in \tilde{V}\\ &\alpha_2(\gamma(\psi', v')) \subseteq v' && \psi' \in \tilde{\Psi}, \quad v' \in \tilde{V}\\ &v \subseteq \gamma(\alpha_1(v), \alpha_2(v)) && v \in \mathcal{P}(V)\end{aligned}$$

4.3 Closure analysis

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

$$\begin{aligned}&\tilde{\mathbf{E}}[\![e]\!] : \tilde{\Psi}^n \to \tilde{V}^k \to \tilde{V} \times (\tilde{V}^k)^n\\ &\tilde{\mathbf{U}}[\![p]\!] : (\tilde{V}^k)^n \to (\tilde{V}^k \times \tilde{V})^n\end{aligned}$$

$$\begin{aligned}
\tilde{\mathbf{E}}[\![x_i]\!]\psi\rho &= && \langle \rho_i, \emptyset^n\rangle\\
\tilde{\mathbf{E}}[\![f_i]\!]\psi\rho &= && \langle \{[i, 0]\}, \emptyset^n\rangle\\
\tilde{\mathbf{E}}[\![a_j(e_1, \ldots, e_k)]\!]\psi\rho &= && \textbf{let } \langle d_i, c_i\rangle = \tilde{\mathbf{E}}[\![e_i]\!]\psi\rho \textbf{ for } i = 1, \ldots, k \textbf{ in}\\
&&& \textbf{if } \underline{atom} \notin d_i \textbf{ for some } i \textbf{ then } \langle \emptyset, c_1 \sqcup \cdots \sqcup c_k\rangle\\
&&& \textbf{else } (\{\underline{atom}\}, c_1 \sqcup \cdots \sqcup c_k)\\
\tilde{\mathbf{E}}[\![e_1(e_2)]\!]\psi\rho &= && \textbf{let } \langle d_1, c_1\rangle = \tilde{\mathbf{E}}[\![e_1]\!]\psi\rho \textbf{ and}\\
&&& \quad \langle d_2, c_2\rangle = \tilde{\mathbf{E}}[\![e_2]\!]\psi\rho \textbf{ in}\\
&&& (\textstyle\bigcup_{[i,j]\in d_1} \{\textbf{if } j = k - 1 \textbf{ then } lookup'_i(\psi) \textbf{ else } [i, j + 1]\},\\
&&& \textstyle\bigsqcup_{[i,j]\in d_1} only_i(\underbrace{\bot, \ldots, \bot}_{j}, d2, \bot, \ldots))
\end{aligned}$$

$$\begin{aligned}
\tilde{\mathbf{E}}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\psi\rho =&\\
&\textbf{let } \langle d_1, c_1\rangle = \tilde{\mathbf{E}}[\![e_1]\!]\psi\rho\\
&\quad \langle d_2, c_2\rangle = \tilde{\mathbf{E}}[\![e_2]\!]\psi\rho\\
&\quad \langle d_3, c_3\rangle = \tilde{\mathbf{E}}[\![e_3]\!]\psi\rho \textbf{ in}\\
&\textbf{if } \underline{atom} \in d_1 \textbf{ then } \langle d_2 \cup d_3, c_1 \sqcup c_2 \sqcup c_3\rangle\\
&\textbf{else } (\emptyset, c_1)
\end{aligned}$$

$$\tilde{\mathbf{U}}[\![f_1\ x_1 \cdots x_k = e_1 \quad \ldots \quad f_n\ x_1 \cdots x_k = e_n]\!]\tilde{c} =$$
$$\mathbf{fix}\lambda\psi.\, savecalls'(\vec{c}) \sqcup$$
$$\langle map'_1(\mathbf{E}_m[\![e_1]\!]\psi, getcalls'(\psi) \downarrow 1), \ldots,$$
$$map'_1(\mathbf{E}_m[\![e_n]\!]\psi, getcalls'(\psi) \downarrow n) \sqcup$$
$$savecalls'(map'_2(\mathbf{E}_m[\![e_1]\!]\psi, getcalls'(\psi) \downarrow 1)) \sqcup \cdots \sqcup$$
$$savecalls'(map'_2(\mathbf{E}_m[\![e_n]\!]\psi, getcalls'(\psi) \downarrow n))$$

with

$$lookup'_i(\psi) = \mathbf{let}\ \langle\langle v_1, \ldots, v_k\rangle, v_r\rangle = \psi \downarrow i\ \mathbf{in}\ v_r$$
$$only'_i(\vec{v}) = \langle \underbrace{\emptyset^k, \ldots, \emptyset^k}_{i-1}, \vec{v}, \emptyset^k, \ldots\rangle$$
$$map'_1(f, \vec{v}) = \langle \vec{v}, f(\vec{v}) \downarrow 1\rangle$$
$$map'_2(f, \vec{v}) = f(\vec{v}) \downarrow 2$$
$$savecalls'(\langle \vec{v}_1, \ldots, \vec{v}_n\rangle) = \langle\langle \vec{v}_1, \emptyset\rangle, \ldots, \langle \vec{v}_n, \emptyset\rangle\rangle$$
$$getcalls'(\langle\langle \vec{v}_1, r_1\rangle, \ldots, \langle \vec{v}_n, r_n\rangle\rangle) = \langle \vec{v}_1, \ldots, \vec{v}_n\rangle$$

4.4 Example

Consider the factorial function written in continuation passing style:

$$\begin{aligned} f\ n\ k &= \mathbf{if}\ n = 0\ \mathbf{then}\ k\ 1\ \mathbf{else}\ f\ (n-1)\ (m\ n\ k) \\ m\ n\ k\ x &= k(n * x) \\ id\ x &= x \\ fac\ x &= f\ x\ id \end{aligned}$$

The initial call description states that only the factorial function fac can be called externally and that its argument will be a base value. We will in this example use function names for function identification in closures. The initial call will then be: $only'_{fac}(\langle \underline{atom}\rangle)$ The closure analysis computes the following function environment:

$$\begin{aligned} f: &\quad \langle\{\underline{atom}\}, \{[id, 0], [m, 2]\}, \{\underline{atom}\}\rangle \\ m: &\quad \langle\{\underline{atom}\}, \{[id, 0], [m, 2]\}, \{\underline{atom}\}, \{\underline{atom}\}\rangle \\ id: &\quad \langle\{\underline{atom}\}, \{\underline{atom}\}\rangle \\ fac: &\quad \langle\{\underline{atom}\}, \{\underline{atom}\}\rangle \end{aligned}$$

The second argument to the functions f and m will either be id with no further arguments or k with two arguments.

4.5 Example

Consider the tautology function from the introduction

$$\begin{aligned} f &= taut\ g\ 2 \\ g\ x\ y &= x \wedge \neg y \\ taut\ h\ n &= h, \quad \mathbf{if}\ n = 0 \\ &= taut\ (h\ true)\ (n-1) \wedge taut\ (h\ false)\ (n-1), \quad \mathbf{otherwise} \end{aligned}$$

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

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

4.6 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. More formally we extend the abstraction function to call descriptions α_c and function environments α_ψ :

$$
\begin{array}{l}
\alpha_c : C^n \to (\tilde{V}^k)^n \qquad \text{or} \quad (\mathcal{P}(V^k))^n \to (\tilde{V}^k)^n \\
\alpha_c(\langle c_1,\ldots,c_n\rangle) = \langle\langle\alpha_2(\{\rho_1 \mid \rho \in c_1\}),\ldots,\alpha_2(\{\rho_k \mid \rho \in c_1\}\rangle,\ldots\rangle
\end{array}
$$

$$
\begin{array}{l}
\alpha_\psi : \Psi^n \to \tilde{\Psi} \qquad \text{or} \quad (\mathcal{P}(V^k \times V_\perp))^n \to (\tilde{V}^k \times \tilde{V})^n \\
\alpha_\psi(\langle\psi_1,\ldots,\psi_n\rangle) = \\
\quad \langle\langle\alpha_2(\{\rho_1 \mid \langle\rho,r\rangle \in \psi_1\}),\ldots,\alpha_2(\{\rho_k \mid \langle\rho,r\rangle \in \psi_1\}\rangle, \\
\qquad\qquad\qquad \alpha_2(\{\rho_k \mid \langle\rho,r\rangle \in \psi_1 \wedge r \neq \perp\}\rangle,\ldots\rangle \\
\quad \sqcup \ \bigsqcup_{\langle\langle v_1,\ldots,v_k\rangle,v_r\rangle\in\psi_i}(\alpha_1(v_1)\sqcup\cdots\sqcup\alpha_1(v_k)\sqcup\alpha_1(v_r))
\end{array}
$$

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.

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

Local safety Given function and parameter environments

$$
\begin{array}{ll}
\rho \in V^k & \rho' \in \tilde{V}^k \\
\psi \in \Psi^n & \psi' \in \tilde{\Psi}^n
\end{array}
$$

where

$$
\begin{array}{l}
\alpha_2(\rho_i) \sqsubseteq \rho'_i \qquad i \in \{1,\ldots,k\} \\
\alpha_\psi(\psi) \sqcup \alpha_1(\rho_1) \sqcup \cdots \sqcup \alpha_1(\rho_k) \sqsubseteq \psi'
\end{array}
$$

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

$$
\begin{array}{l}
\alpha_2(\{\mathbf{E}_m[\![e]\!]\psi\rho \downarrow 1\} \setminus \{\perp\}) \subseteq \tilde{\mathbf{E}}[\![e]\!]\psi'\rho' \downarrow 1 \\
\alpha_c(\mathbf{E}_m[\![e]\!]\psi\rho \downarrow 2) \sqsubseteq \tilde{\mathbf{E}}[\![e]\!]\psi'\rho' \downarrow 2
\end{array}
$$

for any expression e.

The safety proof is based on structural induction over expressions and fixpoint induction. The proof may be found in the extended version of this paper.

5 Conclusion

The paper describes a minimal function graph semantics for a higher order strict functional language. The semantics is related to a closure semantics for the 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.

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

References

1. A Bondorf. *Similix user manual.* Tech. Report. 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 Formal Description of Programming Concepts (E J Neuhold, ed.). North-Holland, 1978.
4. 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.
5. A Mycroft and M Rosendahl. *Minimal Function Graphs are not instrumented.* In WSA'92, Bordeaux, France, pp. 60–67. Bigre. Irisa Rennes, France, Sept., 1992.
6. P Sestoft. *Replacing Function Parameters by Global Variables.* M.Sc. Thesis 88-7-2. DIKU, Univ. of Copenhagen, Denmark, Oct., 1988.
7. O Shivers. *Control flow analysis in Scheme.* In SIGPLAN '88 Conference on PLDI, Atlanta, Georgia, pp. 164–174. Volume 23(7) of ACM SIGPLAN Not., July, 1988.