

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/220997870>

# Data Flow Analysis of Applicative Programs Using Minimal Function Graphs.

Conference Paper · January 1986

DOI: 10.1145/512644.512672 · Source: DBLP

---

CITATIONS

137

READS

56

2 authors, including:



[Neil D. Jones](#)

University of Copenhagen

151 PUBLICATIONS 7,532 CITATIONS

[SEE PROFILE](#)

# The Journal of Functional and Logic Programming

*The MIT Press*

Volume 1997, Article 2  
*27 February, 1997*

ISSN 1080-5230. MIT Press Journals, 55 Hayward St., Cambridge, MA 02142, USA; (617)253-2889; *journals-orders@mit.edu, journals-info@mit.edu*. Published one article at a time in L<sup>A</sup>T<sub>E</sub>X source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://www.cs.tu-berlin.de/journal/jflp/*
- *http://mitpress.mit.edu/jrnls-catalog/jflp.html*
- *gopher.mit.edu*
- anonymous *ftp* at *mitpress.mit.edu*

©1997 Massachusetts Institute of Technology. Subscribers are licensed to use journal articles in a variety of ways, limited only as required to insure fair attribution to authors and the journal, and to prohibit use in a competing commercial product. See the journal's World Wide Web site for further details. Address inquiries to the Subsidiary Rights Manager, MIT Press Journals; (617)253-2864; *journals-rights@mit.edu*.

*The Journal of Functional and Logic Programming* is a peer-reviewed and electronically published scholarly journal that covers a broad scope of topics from functional and logic programming. In particular, it focuses on the integration of the functional and the logic paradigms as well as their common foundations.

*Editor-in-Chief:* G. Levi

<i>Editorial Board:</i>	H. Aït-Kaci	L. Augustsson
	Ch. Brzozka	J. Darlington
	Y. Guo	M. Hagiya
	M. Hanus	T. Ida
	J. Jaffar	B. Jayaraman
	M. Köhler*	A. Krall*
	H. Kuchen*	J. Launchbury
	J. Lloyd	A. Middeldorp
	D. Miller	J. J. Moreno-Navarro
	L. Naish	M. J. O'Donnell
	P. Padawitz	C. Palamidessi
	F. Pfenning	D. Plaisted
	R. Plasmeijer	U. Reddy
	M. Rodríguez-Artalejo	F. Silbermann
	P. Van Hentenryck	D. S. Warren

\* Area Editor

<i>Executive Board:</i>	M. M. T. Chakravarty	A. Hallmann
	H. C. R. Lock	R. Loogen
	A. Mück	

*Electronic Mail:* [jflp.request@ls5.informatik.uni-dortmund.de](mailto:jflp.request@ls5.informatik.uni-dortmund.de)

# Higher-Order Minimal Function Graphs

Neil D. Jones      Mads Rosendahl

27 February, 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.

## 1 Introduction

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

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

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

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

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

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 (e.g.,  $\perp, \top$ ). This approach is taken by Burn, Hankin, and Abramsky [BHA86] for strictness analysis, and may be used for other *compositional* analyses where program parts can be analyzed independently of the contexts in which they occur.

In contrast, the goal of a *top-down* analysis is to describe the effects of applying the program to a given set of input values, and therefore this type of analysis is context dependent. A top-down 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  $IO_f = \{(a, f(a)), (b, f(b)), \dots\}$ , so a set of needed function values corresponds to a perhaps proper subset of  $IO_f$ . This leads to an approach described by Jones and Mycroft [JM86]. In the minimal function-graph (MFG) approach to semantics, the idea is to describe each function by the smallest set of argument-result pairs that is 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

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

and the informal definition may be

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

By this approach, one analyzes a program by approximating its MFG semantics, as outlined by Jones and Mycroft [JM86] and developed in several places since.

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

Our method is again semantically based, but on a more operational level than, for example, that of Burn, Hankin, and Abramsky [BHA86]. Three steps are involved:

1. Define a closure-based semantics for higher-order programs (in a now traditional way).
2. Define a minimal function graph variant of this, which collects only reachable function arguments and results.

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

In this paper, we present 1 and 2 in some detail, and sketch an application as in 3.

## 2 Language

Consider a small language that is 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:

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

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

$x_i$	Parameters
$f_i$	Function names
$a_i(e_1, \dots, e_k)$	Basic operations
$e_1(e_2)$	Application
<b>if</b> $e_1$ <b>then</b> $e_2$ <b>else</b> $e_3$	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  $f_1, \dots, f_n$ , parameters  $x_1, \dots, x_k$ , and basic operations  $a_1, a_2, \dots$ .

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

**Example 1** As an example of how to use the curried style in the language, consider the following program.

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

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

**End of Example 1**  $\square$

## 3 Closure Semantics

We have not specified which basic operations and data types the language should contain. The details are not important, as long as the underlying data type contains the truth values. We will therefore assume that the language has an underlying data type represented with the set *Base*, and that the basic operations  $a_1, a_2, \dots$  have standard meanings as functions  $\underline{a}_i : \text{Base}^k \rightarrow \text{Base}_\perp$ . 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).

### 3.1 Closures

The semantics presented below is based on “closures,” so a partially applied function will be represented as a closure or tuple  $[i, v_1, \dots, v_j]$  of the function identification *i* for the function name  $f_i$  and the so-far computed arguments  $(v_1, \dots, v_j)$ . Only when all arguments are provided, i.e., when  $j = k$ , will the expression  $e_i$  defining the *i*<sup>th</sup> 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:

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

Here we use the numbers  $1, \dots, n$  instead of the function names, to remove the syntactic information from the semantic domains. As an example,  $[i, \epsilon]$ , with  $\epsilon$  as the empty sequence, denotes the  $i^{\text{th}}$  function before being applied to any arguments. We will use square brackets to construct and denote closures in expressions such as  $[i, v_1, \dots, v_\ell]$ . 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.

### 3.2 Fixpoint Semantics with Closures

We have the following semantic domains:

$$\begin{array}{ll} V = \text{Base} \cup (\{1, \dots, n\} \times V^*) & v, w \in V_\perp, \rho \in V^k \\ \Phi = V^k \rightarrow V_\perp & \phi \in \Phi^n \end{array}$$

And we have the following semantic functions:

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

where the definitions from Figure 1 are assumed.

### 3.3 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.

$$\begin{aligned}
\mathbf{E}_c[x_i]\phi\rho &= \rho \downarrow i \\
\mathbf{E}_c[f_i]\phi\rho &= [i, \epsilon] \\
\mathbf{E}_c[a_j(e_1, \dots, e_k)]\phi\rho &= \\
&\quad \text{let } v_i = \mathbf{E}_c[e_i]\phi\rho \text{ for } i = 1, \dots, k \text{ in} \\
&\quad \text{if } v_i \notin \text{Base} \text{ for some } i \text{ then } \perp \text{ else } \underline{a}_j(v_1, \dots, v_k) \\
\mathbf{E}_c[e_1(e_2)]\phi\rho &= \\
&\quad \text{let } w_1 = \mathbf{E}_c[e_1]\phi\rho \text{ and } w_2 = \mathbf{E}_c[e_2]\phi\rho \text{ in} \\
&\quad \text{if } w_1 = \perp \text{ or } w_2 = \perp \text{ or } w_1 \in \text{Base} \text{ then } \perp \text{ else} \\
&\quad \text{let } [i, v_1, \dots, v_\ell] = w_1 \text{ in} \\
&\quad \text{if } \ell + 1 < k \text{ then } [i, v_1, \dots, v_\ell, w_2] \\
&\quad \text{else } \phi_i(v_1, \dots, v_\ell, w_2) \\
\mathbf{E}_c[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\phi\rho &= \\
&\quad \text{if } \mathbf{E}_c[e_1]\phi\rho = \text{true} \text{ then } \mathbf{E}_c[e_2]\phi\rho \text{ else} \\
&\quad \text{if } \mathbf{E}_c[e_1]\phi\rho = \text{false} \text{ then } \mathbf{E}_c[e_3]\phi\rho \text{ else } \perp \\
\mathbf{U}_c[f_1 \ x_1 \cdots x_k = e_1 \ \dots \ f_n \ x_1 \cdots x_k = e_n] &= \\
&\quad \text{fix } \lambda\phi.\langle \mathbf{E}_c[e_1]\phi, \dots, \mathbf{E}_c[e_n]\phi \rangle
\end{aligned}$$

Figure 1: Definitions of semantic functions for the closure semantics.

## 4 The 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 that are sufficient to include all calls resulting from applying the program to an *initial call description*. This is of the form  $\langle c_1, \dots, c_n \rangle$  where each  $c_i$  is the set of arguments with which  $f_i$  is called externally.

### 4.1 The 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 also 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. This may be described by a power set

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

A connection between  $\Psi$  and  $C$  can be given with the following two functions:

$$\begin{aligned} \text{getcalls} &: \Psi \rightarrow C \\ \text{getcalls}(\psi) &= \{\vec{v} \mid \langle \vec{v}, r \rangle \in \psi\} \\ \text{savecalls} &: C \rightarrow \Psi \\ \text{savecalls}(c) &= \{\langle \vec{v}, \perp \rangle \mid \vec{v} \in c\} \end{aligned}$$

If  $C$  is ordered with set inclusion and function denotations  $\Psi$  are ordered with the ordering

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

then  $C$  may be seen as an abstraction of the domain of function denotations  $\Psi$ . Clearly,  $getcalls \circ savecalls$  is the identity on  $C$  and

$$\text{for all } \psi \in \Psi, savecalls(getcalls(\psi)) \sqsubseteq \psi$$

Component-wise, the functions  $getcalls$  and  $savecalls$  are extended to  $\Psi^n$  and  $C^n$ .

## 4.2 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 the function arguments that are needed to do the evaluation.

We have the following semantic domains:

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

And we have the following semantic functions:

$$\begin{aligned}\mathbf{E}_m[e] &: \Psi^n \rightarrow V^k \rightarrow V_\perp \times C^n \\ \mathbf{U}_m[p] &: C^n \rightarrow \Psi^n\end{aligned}$$

The function  $\mathbf{E}_m$  is defined in Figure 2. The function  $\mathbf{U}_m$  maps program-input needs to an  $n$ -tuple of minimal function graphs, one for each  $f_i$ . It is defined in Figure 3. The definition of  $\mathbf{U}_m$  states that the function environment  $\psi$  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

$$(savecalls(map_2(\mathbf{E}_m[e_i]\psi, getcalls(\psi) \downarrow i)))$$

$$\begin{aligned}
\mathbf{E}_m[x_i]\psi\rho &= \langle \rho \downarrow i, nocalls \rangle \\
\mathbf{E}_m[f_i]\psi\rho &= \langle [i, \epsilon], nocalls \rangle \\
\mathbf{E}_m[a_j(e_1, \dots, e_k)]\psi\rho &= \\
&\quad \text{let } \langle w_i, \vec{c}_i \rangle = \mathbf{E}_m[e_i]\psi\rho \text{ for } i = 1, \dots, k \text{ in} \\
&\quad \text{if } w_i \notin Base \text{ for some } i \text{ then } \langle \perp, \vec{c}_1 \sqcup \dots \sqcup \vec{c}_k \rangle \\
&\quad \text{else } \langle \underline{a}_j(w_1, \dots, w_k), \vec{c}_1 \sqcup \dots \sqcup \vec{c}_k \rangle \\
\mathbf{E}_m[e_1(e_2)]\psi\rho &= \\
&\quad \text{let } \langle w_1, \vec{c}_1 \rangle = \mathbf{E}_m[e_1]\psi\rho \text{ and} \\
&\quad \langle w_2, \vec{c}_2 \rangle = \mathbf{E}_m[e_2]\psi\rho \text{ in} \\
&\quad \text{if } w_1 = \perp \text{ or } w_2 = \perp \text{ or } w_1 \in Base \text{ then } \langle \perp, \vec{c}_1 \sqcup \vec{c}_2 \rangle \\
&\quad \text{else let } [i, v_1, \dots, v_\ell] = w_1 \text{ in} \\
&\quad \text{if } \ell + 1 < k \text{ then } \langle [i, v_1, \dots, v_\ell, w_2], \vec{c}_1 \sqcup \vec{c}_2 \rangle \\
&\quad \text{else } \langle \text{lookup}_i(\langle v_1, \dots, v_\ell, w_2 \rangle, \psi), \\
&\quad \quad \vec{c}_1 \sqcup \vec{c}_2 \sqcup \text{only}_i(\langle v_1, \dots, v_\ell, w_2 \rangle) \rangle \\
\mathbf{E}_m[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\psi\rho &= \\
&\quad \text{let } \langle w_1, \vec{c}_1 \rangle = \mathbf{E}_m[e_1]\psi\rho \\
&\quad \langle w_2, \vec{c}_2 \rangle = \mathbf{E}_m[e_2]\psi\rho \\
&\quad \langle w_3, \vec{c}_3 \rangle = \mathbf{E}_m[e_3]\psi\rho \text{ in} \\
&\quad \text{if } w_1 = \text{true} \text{ then } \langle w_2, \vec{c}_1 \sqcup \vec{c}_2 \rangle \\
&\quad \text{else if } w_1 = \text{false} \text{ then } \langle w_3, \vec{c}_1 \sqcup \vec{c}_3 \rangle \text{ else } \langle \perp, \vec{c}_1 \rangle
\end{aligned}$$

with

$$\begin{aligned}
\text{lookup} &: V^k \times \Psi \rightarrow V_\perp \\
\text{lookup}_i(\vec{v}, \psi) &= \bigsqcup \{r \mid \langle \vec{v}, r \rangle \in (\psi \downarrow i)\} \\
\text{only}_i &: V^k \rightarrow C^n \\
\text{only}_i(\vec{v}) &= \underbrace{\langle \emptyset, \dots, \emptyset \rangle}_{i-1}, \{ \vec{v} \}, \emptyset, \dots, \emptyset \\
\text{nocalls} &= \underbrace{\langle \emptyset, \dots, \emptyset \rangle}_n
\end{aligned}$$

Figure 2: Definition of the function  $\mathbf{E}_m$ .

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

with

$$\begin{aligned}
\text{map}_1(f, s) &= \{\langle \vec{v}, r \rangle \mid \vec{v} \in s \wedge f(\vec{v}) = \langle r, \_ \rangle\} \\
\text{map}_2(f, s) &= \bigsqcup \{c \mid \vec{v} \in s \wedge f(\vec{v}) = \langle \_, c \rangle\}
\end{aligned}$$

Figure 3: Definition of the function  $\mathbf{U}_m$ .

### 4.3 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

$$\forall p, \vec{v}, j. (\mathbf{U}_c[p] \downarrow j) \vec{v} = \text{lookup}_j(\vec{v}, \mathbf{U}_m[p] \text{only}_j(\vec{v}))$$

It may be done as for the first-order case by Mycroft and Rosendahl [MR92] where also the stronger condition is proved that the minimal function-graph semantics only contains the needed computations.

### 4.4 Comment

We have essentially used the same ordering on the domain  $\Psi$  as one would have used on the function domain  $V^k \rightarrow V_\perp$ ; it might have been tempting to use a function domain rather than a power-set construction with an unusual ordering. It is, however, important to distinguish 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  $V^k \rightarrow V_\perp$ . A different description would be to use a function domain with a double-lifted result domain  $V^k \rightarrow V_{!\perp}$  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 [JM86], whereas our approach is closer to Cousots's description of chaotic fixpoint iteration [CC78].

## 5 Closure Analysis

As an example of how one may use this semantics in program analysis, we will sketch a closure analysis for the language. For each use of a functional expression, closure analysis will 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 [Ses88]. 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 [Bon93], and was independently rediscovered by Shivers [Shi88].

### 5.1 Abstract Closures

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

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

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  $i^{\text{th}}$  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

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

## 5.2 Relating Abstract and Concrete Closures

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

$$\begin{aligned}\alpha_1 : V_{\perp} &\rightarrow \tilde{V} \\ \alpha_2 : V_{\perp} &\rightarrow (\tilde{V}^k)^n\end{aligned}$$

and definitions

$$\begin{aligned}\alpha_1([i, v_1, \dots, v_j]) &= \{[i, j]\} \\ \alpha_1(c) &= \{\underline{\text{atom}}\} & c \in \text{Base} \\ \alpha_1(\perp) &= \emptyset \\ \alpha_2([i, v_1, \dots, v_j]) &= \\ &\quad \langle \underbrace{\langle \emptyset, \dots, \emptyset \rangle}_{i-1}, \dots, \langle \alpha_1(v_1), \dots, \alpha_1(v_j), \emptyset, \dots \rangle, \langle \emptyset, \dots, \emptyset \rangle, \dots \rangle \\ \alpha_2(c) &= \langle \langle \emptyset, \dots, \emptyset \rangle, \dots \rangle & c \in \text{Base} \\ \alpha_2(\perp) &= \langle \langle \emptyset, \dots, \emptyset \rangle, \dots \rangle\end{aligned}$$

The abstraction functions  $\alpha_1$  and  $\alpha_2$  may be extended to  $\mathcal{P}(V)$  as the  $\cup$  and  $\sqcup$ -closure, respectively, so that we have the functionalities

$$\begin{aligned}\alpha_1 : \mathcal{P}(V) &\rightarrow \tilde{V} \\ \alpha_2 : \mathcal{P}(V) &\rightarrow (\tilde{V}^k)^n\end{aligned}$$

## 5.3 Closure Analysis

The closure analysis should map a program into an abstract function environment that safely approximates the minimal function-graph interpretation. We have the following semantic domains:

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

And we have the following semantic functions:

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

The definitions of  $\tilde{\mathbf{E}}$  and  $\tilde{\mathbf{U}}$  are given in the Figures 4 and 5, respectively.

**Example 2** Consider the tautology function from the introduction

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

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

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

**End of Example 2**  $\square$

## 5.4 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  $\tilde{\mathbf{U}}$ .

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.

$$\begin{aligned}\alpha_c &: C^n \rightarrow (\tilde{V}^k)^n \\ \alpha_c(\langle c_1, \dots, c_n \rangle) &= \langle \langle \alpha_1(\{\rho \downarrow 1 \mid \rho \in c_1\}), \dots, \alpha_1(\{\rho \downarrow k \mid \rho \in c_1\}) \rangle, \dots \rangle \\ \alpha_\psi &: \Psi^n \rightarrow \tilde{\Psi}^n \\ \alpha_\psi(\psi) &= \widetilde{\text{savecalls}}(\alpha_c(\text{getcalls}(\psi))) \sqcup \\ &\quad \widetilde{\text{savecalls}}(\bigsqcup_i (\alpha_2(\{r \mid \langle \_, r \rangle \in \psi \downarrow i\} \setminus \{\perp\}))) \\ &\quad \langle \langle \emptyset, \dots, \emptyset, \alpha_1(\{r \mid \langle \_, r \rangle \in \psi \downarrow 1\}) \rangle, \dots \rangle\end{aligned}$$

$$\begin{aligned}
\tilde{\mathbf{E}}[x_i]\tilde{\psi}\tilde{\rho} &= \langle \tilde{\rho} \downarrow i, nocalls \rangle \\
\tilde{\mathbf{E}}[f_i]\tilde{\psi}\tilde{\rho} &= \langle \{[i, 0]\}, nocalls \rangle \\
\tilde{\mathbf{E}}[a_j(e_1, \dots, e_k)]\tilde{\psi}\tilde{\rho} &= \text{let } \langle \tilde{w}_i, \tilde{c}_i \rangle = \tilde{\mathbf{E}}[e_i]\tilde{\psi}\tilde{\rho} \text{ for } i = 1, \dots, k \text{ in} \\
&\quad \text{if } \underline{atom} \notin \tilde{w}_i \text{ for some } i \text{ then } \langle \emptyset, \tilde{c}_1 \sqcup \dots \sqcup \tilde{c}_k \rangle \\
&\quad \text{else } (\{\underline{atom}\}, \tilde{c}_1 \sqcup \dots \sqcup \tilde{c}_k) \\
\tilde{\mathbf{E}}[e_1(e_2)]\tilde{\psi}\tilde{\rho} &= \text{let } \langle \tilde{w}_1, \tilde{c}_1 \rangle = \tilde{\mathbf{E}}[e_1]\tilde{\psi}\tilde{\rho} \text{ and} \\
&\quad \langle \tilde{w}_2, \tilde{c}_2 \rangle = \tilde{\mathbf{E}}[e_2]\tilde{\psi}\tilde{\rho} \text{ in} \\
&\quad \text{if } \tilde{w}_2 = \emptyset \text{ then } \langle \perp, \tilde{c}_1 \sqcup \tilde{c}_2 \rangle \text{ else} \\
&\quad \langle \bigcup_{[i,j] \in \tilde{w}_1} \{ \text{if } j = k-1 \text{ then } \widetilde{lookup}_i(\tilde{\psi}) \\
&\quad \quad \text{else } [i, j+1] \}, \\
&\quad \tilde{c}_1 \sqcup \tilde{c}_2 \sqcup \bigsqcup_{[i,j] \in \tilde{w}_1} \widetilde{only}_i(\langle \underbrace{\emptyset, \dots, \emptyset}_j, \tilde{w}_2, \emptyset, \dots \rangle) \rangle \\
\tilde{\mathbf{E}}[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\tilde{\psi}\tilde{\rho} &= \\
&\quad \text{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} \text{ in} \\
&\quad \text{if } \underline{atom} \in \tilde{w}_1 \text{ then } \langle \tilde{w}_2 \cup \tilde{w}_3, \tilde{c}_1 \sqcup \tilde{c}_2 \sqcup \tilde{c}_3 \rangle \\
&\quad \text{else } (\emptyset, \tilde{c}_1)
\end{aligned}$$

with

$$\begin{aligned}
\widetilde{lookup}_i(\tilde{\psi}) &= \text{let } \langle \langle \tilde{v}_1, \dots, \tilde{v}_k \rangle, \tilde{v}_r \rangle = \tilde{\psi} \downarrow i \text{ in } \tilde{v}_r \\
\widetilde{only}_i(\tilde{v}) &= \langle \underbrace{\text{noargs}, \dots, \text{noargs}}_{i-1}, \tilde{v}, \text{noargs}, \dots \rangle \\
\text{noargs} &= \langle \underbrace{\emptyset, \dots, \emptyset}_k \rangle \\
\text{nocalls} &= \langle \underbrace{\text{noargs}, \dots, \text{noargs}}_n \rangle
\end{aligned}$$

Figure 4: Definition of the function  $\tilde{\mathbf{E}}$ .

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

with

$$\begin{aligned}
\widetilde{\text{map}}_1(f, \tilde{v}) &= \langle \tilde{v}, f(\tilde{v}) \downarrow 1 \rangle \\
\widetilde{\text{map}}_2(f, \tilde{v}) &= f(\tilde{v}) \downarrow 2 \\
\widetilde{\text{savecalls}}(\langle \tilde{v}_1, \dots, \tilde{v}_n \rangle) &= \langle \langle \tilde{v}_1, \emptyset \rangle, \dots, \langle \tilde{v}_n, \emptyset \rangle \rangle \\
\widetilde{\text{getcalls}}(\langle \langle \tilde{v}_1, \tilde{r}_1 \rangle, \dots, \langle \tilde{v}_n, \tilde{r}_n \rangle \rangle) &= \langle \tilde{v}_1, \dots, \tilde{v}_n \rangle
\end{aligned}$$

Figure 5: Definition of the function  $\tilde{\mathbf{U}}$ .

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.

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.

**Theorem 1 (Local Safety)** *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

$$\begin{aligned}
\alpha_1(\rho \downarrow i) \subseteq \tilde{\rho} \downarrow i &\quad \forall i \in \{1, \dots, k\} \\
(\bigsqcup_i \alpha_2(\rho \downarrow i)) \subseteq \widetilde{\text{getcalls}}(\tilde{\psi}) &\quad \forall i \in \{1, \dots, k\} \\
\alpha_\psi(\psi) \sqsubseteq \tilde{\psi}
\end{aligned}$$

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

$$\begin{aligned}\alpha_1(\{\mathbf{E}_m[e]\psi\rho\downarrow 1\} \setminus \{\perp\}) &\subseteq \tilde{\mathbf{E}}[e]\tilde{\psi}\tilde{\rho}\downarrow 1 \quad \wedge \\ \alpha_c(\mathbf{E}_m[e]\psi\rho\downarrow 2) &\sqsubseteq \tilde{\mathbf{E}}[e]\tilde{\psi}\tilde{\rho}\downarrow 2 \sqcup \widetilde{\text{getcalls}}(\tilde{\psi})\end{aligned}$$

for any expression  $e$ .

The second requirement for environments is that all partial function applications that are 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 of Theorem 1 (Local Safety)** The proof is by structural induction over expressions. The proof for the first component of the functions is fairly straightforward. 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  $\alpha_1(\perp) \subseteq \emptyset$  and  $\alpha_1(\underline{a}_j(w_1, \dots, w_k)) \subseteq \{\underline{\text{atom}}\}$ . For a function application, let  $w_1 = [i, v_1, \dots, v_j]$  where  $\alpha_1(w_1) = [i, j]$ . It then follows that

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

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  $\alpha_c(\text{nocalls}) = \text{nocalls}$ .

For function applications, it involves proving the relation

$$\begin{aligned}\alpha_c(\text{only}_i(\langle v_1, \dots, v_\ell, w_2 \rangle)) &\sqsubseteq \\ \widetilde{\text{only}}_i(\langle \emptyset, \dots, \emptyset, \tilde{w}_2, \emptyset, \dots \rangle) &\sqcup c_1 \sqcup c_2 \sqcup \widetilde{\text{getcalls}}(\tilde{\psi})\end{aligned}$$

where the closure  $[i, v_1, \dots, v_\ell]$  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 guarantees the relations. In the third case there is nothing to prove since  $\ell$  is zero, and in the fourth case it follows from the structural induction over expressions.

**Proof of Theorem 1**  $\square$

**Theorem 2 (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:*

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

**Proof of Theorem 2 (Global Safety)** Define the functions

$$\begin{aligned} F_c(\psi) &= \text{savecalls}(\tilde{c}) \sqcup \\ &\quad \langle \text{map}_1(\mathbf{E}_m[e_1]\psi, \text{getcalls}(\psi) \downarrow 1), \dots, \\ &\quad \text{map}_1(\mathbf{E}_m[e_n]\psi, \text{getcalls}(\psi) \downarrow n) \rangle \sqcup \\ &\quad \text{savecalls}(\text{map}_2(\mathbf{E}_m[e_1]\psi, \text{getcalls}(\psi) \downarrow 1)) \sqcup \dots \sqcup \\ &\quad \text{savecalls}(\text{map}_2(\mathbf{E}_m[e_n]\psi, \text{getcalls}(\psi) \downarrow n)) \\ \tilde{F}_{\tilde{c}}(\tilde{\psi}) &= \widetilde{\text{savecalls}}(\tilde{c}) \sqcup \\ &\quad \langle \widetilde{\text{map}}_1(\tilde{\mathbf{E}}[e_1]\tilde{\psi}, \widetilde{\text{getcalls}}(\tilde{\psi}) \downarrow 1), \dots, \\ &\quad \widetilde{\text{map}}_1(\tilde{\mathbf{E}}[e_n]\tilde{\psi}, \widetilde{\text{getcalls}}(\tilde{\psi}) \downarrow n) \rangle \sqcup \\ &\quad \widetilde{\text{savecalls}}(\widetilde{\text{map}}_2(\tilde{\mathbf{E}}[e_1]\tilde{\psi}, \widetilde{\text{getcalls}}(\tilde{\psi}) \downarrow 1)) \sqcup \dots \sqcup \\ &\quad \widetilde{\text{savecalls}}(\widetilde{\text{map}}_2(\tilde{\mathbf{E}}[e_n]\tilde{\psi}, \widetilde{\text{getcalls}}(\tilde{\psi}) \downarrow n)) \end{aligned}$$

The proof will be based on fixpoint induction since

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

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  $\psi$  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

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

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

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

**Proof of Theorem 2**  $\square$

## 6 Conclusion

This 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.

## 7 Acknowledgments

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

## References

- [BHA86] G. L. Burn, C. L. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.
- [Bon93] Anders Bondorf. Similix user manual. Technical report, DIKU, University of Copenhagen, Denmark, 1993.
- [CC78] 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, Amsterdam, The Netherlands, 1978.
- [JM86] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, pages 296–306, New York, NY, January 1986. ACM Press.
- [MR92] Alan Mycroft and Mads Rosendahl. Minimal function graphs are not instrumented. In *Workshop on Static Analysis, Bordeaux, France*, volume 81–82 of *Bigre*, pages 60–67. IRISA, Campus de Beaulieu, F-35042 Rennes cedex, France, September 1992.

- [Ses88] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988.
- [Shi88] O. Shivers. Control flow analysis in Scheme. In David S. Wise, editor, *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (SIGPLAN '88)*, pages 164–174, New York, NY, July 1988. ACM Press.