The G-Machine as a Representation of Stack Semantics

David Lester,
Programming Research Group, 7–11 Keble Road,
Oxford, OX1 3QD, ENGLAND. *Tel.* +44 865 273869

Introduction

It has been argued [7], that lazy functional programming languages have a simple semantic definition, which allows formal software development methods to be easily used. If the object of this exercise is seen to be the production of a working program, then the implementation should also be correct before the system may be said to meet its specification.

Recent work has shown that such languages can now be efficiently implemented using graph reduction. Because of its concise definition this paper concentrates on Augustsson and Johnsson's G-machine [1] and [2]. Unfortunately, efficiency considerations lead to a complicated implementation, the correctness of which is not immediately obvious.

This paper outlines a proof that an implementation similar to Johnsson's is correct with respect to a stack semantics for a simple lazy functional programming language. This semantics has been shown elsewhere [3] to be congruent to the natural standard semantics for the language. The major difference between the method presented here and that of Johnsson is the use of indirection nodes. The paper also furnishes a proof that Johnsson's alternative strategy is correct.

Π	$\in$	Prog	(Programs)
Δ	$\in$	Defs	(Function Definitions)
Γ	$\in$	Comb	(Combinator Bodies)
E	$\in$	Exp	(Expressions)
Z	$\in$	Int	(Integers)
T	$\in$	Bool	(Truth Values)
I	$\in$	Ide	(Identifiers)

Figure 1: Syntactic Domains

Π	::=	E **where** Δ
Δ	::=	Δ_0 **and** Δ_1 | $I = \Gamma$
Γ	::=	$\lambda I.\Gamma$ | $\lambda I.E$
E	::=	I | Z | T | $E_0\,E_1$

Figure 2: Abstract Syntax

$$
\begin{array}{lcll}
\varepsilon \in \mathbf{E} & = & \mathbf{Z}+\mathbf{T}+\mathbf{L}+\mathbf{F} & \text{(Expression Values)} \\
\mathbf{F} & = & [\mathbf{E}\to\mathbf{E}] & \text{(Function Values)} \\
\ell \in \mathbf{L} & = & (\mathbf{E}\times\mathbf{L})+\{nil\} & \text{(Lists)} \\
o \in \mathbf{O} & = & \mathbf{Z}^* & \text{(Outputs)} \\
\rho \in \mathbf{U} & = & \mathrm{Ide}\to\mathbf{E} & \text{(Environments)}
\end{array}
$$

Figure 3: Value Domains for the Standard Semantics

$$
\begin{array}{lcl}
\mathcal{P} : \mathrm{Prog}\to\mathbf{O} & & \\
\mathcal{P}\,[\![\mathbf{E}\ \textbf{where}\ \Delta]\!] & \equiv & \mathit{print}(\mathcal{E}\,[\![\mathbf{E}]\!]\,(\mathit{fix}(\lambda\rho.\rho_{\mathrm{init}}\oplus\mathcal{D}\,[\![\Delta]\!]\,\rho))) \\
 & & \\
\mathcal{D} : \mathrm{Defs}\to\mathbf{U}\to\mathbf{U} & & \\
\mathcal{D}\,[\![\Delta_0\ \textbf{and}\ \Delta_1]\!]\,\rho & \equiv & \mathcal{D}\,[\![\Delta_0]\!]\,\rho\oplus\mathcal{D}\,[\![\Delta_1]\!]\,\rho \\
\mathcal{D}\,[\![\mathrm{I}=\Gamma]\!]\,\rho & \equiv & \{\mathrm{I}\mapsto\mathcal{F}\,[\![\Gamma]\!]\,\rho\} \\
 & & \\
\mathcal{F} : \mathrm{Comb}\to\mathbf{U}\to\mathbf{E} & & \\
\mathcal{F}\,[\![\lambda\mathrm{I}.\Gamma]\!]\,\rho & \equiv & \lambda\varepsilon.(\mathcal{F}\,[\![\Gamma]\!]\,(\rho\oplus\{\mathrm{I}\mapsto\varepsilon\}))\ \textbf{in}\ \mathbf{E} \\
\mathcal{F}\,[\![\lambda\mathrm{I}.\mathrm{E}]\!]\,\rho & \equiv & \lambda\varepsilon.(\mathcal{E}\,[\![\mathbf{E}]\!]\,(\rho\oplus\{\mathrm{I}\mapsto\varepsilon\}))\ \textbf{in}\ \mathbf{E} \\
 & & \\
\mathcal{E} : \mathrm{Exp}\to\mathbf{U}\to\mathbf{E} & & \\
\mathcal{E}\,[\![\mathrm{I}]\!]\,\rho & \equiv & \mathrm{I}=\mathit{nil}\longrightarrow\mathit{nil}\ \textbf{in}\ \mathbf{E},\ \rho\,[\![\mathrm{I}]\!] \\
\mathcal{E}\,[\![\mathrm{Z}]\!]\,\rho & \equiv & \mathcal{Z}\,[\![\mathrm{Z}]\!] \\
\mathcal{E}\,[\![\mathrm{T}]\!]\,\rho & \equiv & \mathcal{T}\,[\![\mathrm{T}]\!] \\
\mathcal{E}\,[\![\mathrm{E}_0\,\mathrm{E}_1]\!]\,\rho & \equiv & (\varepsilon_0\Vdash\mathbf{F})\longrightarrow(\varepsilon_0\mid\mathbf{F})\varepsilon_1,\ \underline{?} \\
 & & \quad\text{Where}\ \varepsilon_i=\mathcal{E}\,[\![\mathrm{E}_i]\!]\,\rho \\
 & & \\
\mathcal{Z} : \mathrm{Int}\to\mathbf{Z} & & \text{(Not further defined)} \\
 & & \\
\mathcal{T} : \mathrm{Bool}\to\mathbf{T} & & \text{(Not further defined)} \\
 & & \\
\mathit{print}(\varepsilon) & = & (\varepsilon\Vdash\mathbf{Z})\longrightarrow[\varepsilon\mid\mathbf{Z}], \\
 & & (\varepsilon\Vdash\mathbf{T})\longrightarrow[], \\
 & & (\varepsilon\Vdash\mathbf{L})\longrightarrow\mathit{prints}(\varepsilon\mid\mathbf{L}),\ \underline{?} \\
 & & \\
\mathit{prints}(\ell) & = & (\ell=\mathit{nil})\longrightarrow[],\ \mathit{print}(\varepsilon)\mathbin{+\!\!+}\mathit{prints}(\ell') \\
 & & \quad\text{Where}\ (\varepsilon,\ \ell')=\ell\mid(\mathbf{E}\times\mathbf{L})
\end{array}
$$

Figure 4: Standard (or Direct) Semantics

$$
\begin{array}{lcl}
\rho_{\mathrm{init}}\,[\![\mathrm{if}]\!] & = & \underline{\mathit{entry}}3(\underline{\mathit{exit}}3\circ\underline{\mathit{if}}(\underline{\mathit{push}}1)\,(\underline{\mathit{push}}2)\circ\underline{\mathit{eval}}\circ\underline{\mathit{push}}0) \\
\rho_{\mathrm{init}}\,[\![\mathrm{add}]\!] & = & \underline{\mathit{entry}}2(\underline{\mathit{exit}}2\circ\underline{\mathit{add}}\circ\underline{\mathit{eval}}\circ\underline{\mathit{push}}2\circ\underline{\mathit{eval}}\circ\underline{\mathit{push}}0) \\
\rho_{\mathrm{init}}\,[\![\mathrm{eq}]\!] & = & \underline{\mathit{entry}}2(\underline{\mathit{exit}}2\circ\underline{\mathit{eq}}\circ\underline{\mathit{eval}}\circ\underline{\mathit{push}}2\circ\underline{\mathit{eval}}\circ\underline{\mathit{push}}0) \\
\rho_{\mathrm{init}}\,[\![\mathrm{hd}]\!] & = & \underline{\mathit{entry}}1(\underline{\mathit{exit}}1\circ\underline{\mathit{hd}}\circ\underline{\mathit{eval}}\circ\underline{\mathit{push}}0) \\
\rho_{\mathrm{init}}\,[\![\mathrm{null}]\!] & = & \underline{\mathit{entry}}1(\underline{\mathit{exit}}1\circ\underline{\mathit{null}}\circ\underline{\mathit{eval}}\circ\underline{\mathit{push}}0) \\
\rho_{\mathrm{init}}\,[\![\mathrm{cons}]\!] & = & \underline{\mathit{entry}}2(\underline{\mathit{exit}}2\circ\underline{\mathit{cons}}\circ\underline{\mathit{push}}1\circ\underline{\mathit{push}}1)
\end{array}
$$

Figure 5: The Initial Environment ρ_{init}

In Section 1 the language and two alternative semantic definitions are presented. Sections 2 and 3 provide examples of some semantically equivalent pairs of representations for particular forms of expressions, whilst Section 4 concludes the paper.

$$
\begin{array}{lcl}
[] +\!\!+ ys & = & ys \\
(x : xs) +\!\!+ ys & = & x : xs +\!\!+ ys \\
\\
(x : xs) \,!\, 0 & = & x \\
(x : xs) \,!\, (n+1) & = & xs \,!\, n \\
\\
\# [] & = & 0 \\
\# (x : xs) & = & 1 + \# xs \\
\\
take\, 0\, xs & = & [] \\
take\, (n+1)\, (x : xs) & = & x : take\, n\, xs \\
\\
drop\, 0\, xs & = & xs \\
drop\, (n+1)\, (x : xs) & = & drop\, n\, xs \\
\\
map\, f\, [] & = & [] \\
map\, f\, (x : xs) & = & f\, x : map\, f\, xs \\
\\
last\, xs & = & xs \,!\, (\# xs - 1)
\end{array}
$$

Figure 6: Standard List Operations

$$
\begin{array}{lllll}
\sigma & \in & \mathbf{S} & = & \mathbf{O} \times \mathbf{L}^* \times \mathbf{V} \times \mathbf{G} \times \mathbf{U} \times \mathbf{D} \quad \text{(States)} \\
o & \in & \mathbf{O} & = & \mathbf{Z}^* \quad \text{(Output)} \\
\psi & \in & \mathbf{V} & = & (\mathbf{Z} + \mathbf{T})^* \quad \text{(Value Stack)} \\
\gamma & \in & \mathbf{G} & = & [\mathbf{L} \to \mathbf{N}] \quad \text{(Graph Maps)} \\
\nu & \in & \mathbf{N} & = & \mathbf{A} + \mathbf{I} + \mathrm{Ide} + \mathbf{Z} + \mathbf{T} + \mathbf{C} + \{nil\} \quad \text{(Nodes)} \\
 & & \mathbf{A} & = & \mathbf{L} \times \mathbf{L} \quad \text{(Application Nodes)} \\
 & & \mathbf{I} & = & \mathbf{L} \quad \text{(Indirection Nodes)} \\
 & & \mathbf{Z} & = & \{\ldots, -1, 0, 1, \ldots\}_\perp \quad \text{(Integers)} \\
 & & \mathbf{T} & = & \{true, false\}_\perp \quad \text{(Truth Values)} \\
 & & \mathbf{C} & = & \mathbf{L} \times \mathbf{L} \quad \text{(Constructor Nodes)} \\
\rho & \in & \mathbf{U} & = & [\mathrm{Ide} \to \mathbf{K}] \quad \text{(Environments)} \\
\delta & \in & \mathbf{D} & = & \mathbf{L}^{**} \quad \text{(Dumps)} \\
\phi & \in & \mathbf{L}^* & & \quad \text{(Stacks)} \\
\kappa & \in & \mathbf{K} & = & [\mathbf{S} \to \mathbf{S}] \quad \text{(Continuations)} \\
\beta & \in & \mathbf{B} & = & [\mathrm{Ide} \to \mathbf{Z}] \quad \text{(Bindings)} \\
\ell & \in & \mathbf{L} & & \quad \text{(Node Labels)}
\end{array}
$$

Figure 7: Value Domains

1 A Description of the Language

The notation used in this paper broadly follows Stoy [9], to which those readers interested in the mathematics underlying this model are referred. The syntax of the language we wish to describe is given in Figures 1 and 2. Notice that it is simpler than that used by Johnsson in [1], because there are no local definitions. General local definitions require lambda-lifting or some equivalent program transformation. Also, to compile correctly, recursive local definitions need to have definitions of the form $x = y$ eliminated. For these reasons we have omitted local definitions from our analysis.

$$\mathcal{P} : \text{Prog} \to \mathbf{S}$$
$$\mathcal{P}\,[\![\text{E where } \Delta]\!] \equiv (\underline{print} \circ \mathcal{E}\,[\![\text{E}]\!]\,\{\})\,([],\,[],\,[],\,\{\},\,\rho_{\text{init}} \oplus \mathcal{D}\,[\![\Delta]\!],\,[])$$

$$\mathcal{D} : \text{Defs} \to \mathbf{U}$$
$$\mathcal{D}\,[\![\Delta_0 \text{ and } \Delta_1]\!] \equiv \mathcal{D}\,[\![\Delta_0]\!] \oplus \mathcal{D}\,[\![\Delta_1]\!]$$
$$\mathcal{D}\,[\![\text{I} = \Gamma]\!] \equiv \{\text{I} \mapsto \mathcal{F}\,[\![\Gamma]\!]\,\{\}\,0\}$$

$$\mathcal{F} : \text{Comb} \to \mathbf{B} \to \mathbf{Z} \to \mathbf{K}$$
$$\mathcal{F}\,[\![\lambda \text{I}.\Gamma]\!]\,\beta\,n \equiv \mathcal{F}\,[\![\Gamma]\!]\,(\beta \oplus \{\text{I} \mapsto n\})\,(n+1)$$
$$\mathcal{F}\,[\![\lambda \text{I}.\text{E}]\!]\,\beta\,n \equiv \underline{entry}(n+1)\,(\mathcal{R}\,[\![\text{E}]\!]\,(\beta \oplus \{\text{I} \mapsto n\})\,(n+1))$$

$$\mathcal{R} : \text{Exp} \to \mathbf{B} \to \mathbf{Z} \to \mathbf{K}$$
$$\mathcal{R}\,[\![\text{E}]\!]\,\beta\,n \equiv \underline{exit}\,n \circ \mathcal{C}\,[\![\text{E}]\!]\,\beta$$

$$\mathcal{E} : \text{Exp} \to \mathbf{B} \to \mathbf{K}$$
$$\mathcal{E}\,[\![\text{E}]\!]\,\beta \equiv \underline{eval} \circ \mathcal{C}\,[\![\text{E}]\!]\,\beta$$

$$\mathcal{C} : \text{Exp} \to \mathbf{B} \to \mathbf{K}$$
$$\mathcal{C}\,[\![\text{I}]\!]\,\beta \equiv \text{I} = nil \longrightarrow \underline{pushvalue}(nil \text{ in } \mathbf{N})$$
$$\text{I} \in dom(\beta) \longrightarrow \underline{push}(\beta\,[\![\text{I}]\!]),\ \underline{pushvalue}([\![\text{I}]\!] \text{ in } \mathbf{N})$$
$$\mathcal{C}\,[\![\text{Z}]\!]\,\beta \equiv \underline{pushvalue}(\mathcal{Z}\,[\![\text{Z}]\!] \text{ in } \mathbf{N})$$
$$\mathcal{C}\,[\![\text{T}]\!]\,\beta \equiv \underline{pushvalue}(\mathcal{T}\,[\![\text{T}]\!] \text{ in } \mathbf{N})$$
$$\mathcal{C}\,[\![\text{E}_0\,\text{E}_1]\!]\,\beta \equiv \underline{mkap} \circ \mathcal{C}\,[\![\text{E}_0]\!]\,((\lambda x.x+1) \circ \beta) \circ \mathcal{C}\,[\![\text{E}_1]\!]\,\beta$$

$$\mathcal{Z} : \text{Int} \to \mathbf{Z} \qquad \text{(Not further defined)}$$

$$\mathcal{T} : \text{Bool} \to \mathbf{T} \qquad \text{(Not further defined)}$$

Figure 8: Stack Semantics

The standard semantics for the language is given in Figures 3, 4 and 5. Apart from the method for dealing with output this is similar to [10].

Some of the notation differs from that of [9], and so we describe it now. The improper domain elements are $\perp$ which is bottom and $\underline{?}$ which is error. The domain $[\mathbf{A} \to \mathbf{B}]$ denotes the domain of all *computable* functions from $\mathbf{A}$ to $\mathbf{B}$, and elements of this domain can be represented using the λ-notation. The domain $\mathbf{A} = A_\perp$ is the *lifted* domain of the set A. The domain $\mathbf{A}+\mathbf{B}$ is the *seperated-sum* of the domains $\mathbf{A}$ and $\mathbf{B}$, and we have the usual operators $|$ (projection), in (injection) and $\in$ (subdomain test) from [9]. As a shorthand we define $\mathbf{D}^* = (\mathbf{D} \times \mathbf{D}^*) + \{nil\}$. We shall represent elements of $\mathbf{D}^*$ using the SASL notation for lists, and use the standard list operations from Figure 6. The environment combining operator $\oplus$ is defined by $\rho \oplus \{\text{I} \mapsto \varepsilon\} = \lambda x.(x = \text{I} \longrightarrow \varepsilon, \rho x)$, and by its associativity.

We now give the stack semantics for an implementation of this language using graph reduction, and from now on all semantic equations and domains will come from the stack semantics. The value or semantic domains are defined in Figure 7. The stack semantics of Figure 8 and the instructions

$$\underline{pushvalue}\,\nu\,(o, \phi, \psi, \gamma, \rho, \delta) = (o, \ell : \phi, \psi, \gamma \oplus \{\ell \mapsto \nu\}, \rho, \delta) \quad \text{Where } \ell = New(\gamma)$$

$$\underline{push}\,n\,(o, \phi, \psi, \gamma, \rho, \delta) = (o, (\phi\,!\,n) : \phi, \psi, \gamma, \rho, \delta)$$

$$\underline{mkap}\,(o, \ell' : \ell'' : \phi, \psi, \gamma, \rho, \delta) = \underline{pushvalue}\,((\ell', \ell'')\ \mathbf{in}\ \mathbf{N})\,(o, \phi, \psi, \gamma, \rho, \delta)$$

$$\underline{exit}\,n = \underline{unwind} \circ \underline{pop}\,n \circ \underline{update}\,n$$

$$\underline{pop}\,n\,(o, \phi, \psi, \gamma, \rho, \delta) = (o, drop\,n\,\phi, \psi, \gamma, \rho, \delta)$$

$$\underline{update}\,n\,(o, \ell : \phi, \psi, \gamma, \rho, \delta) = (o, \phi, \psi, \gamma \oplus \{(\phi\,!\,n) \mapsto \gamma\,\ell\ \mathbf{in}\ \mathbf{N}\}, \rho, \delta)$$

$$\underline{eval}\,(o, \ell : \phi, \psi, \gamma, \rho, \delta) = (\underline{restore} \circ \underline{unwind})\,(o, [\ell], \psi, \gamma, \rho, \phi : \delta)$$

$$\underline{restore}\,(o, \phi, \psi, \gamma, \rho, \phi' : \delta) = (o, (Elide\,\gamma\,(last\,\phi)) : \phi', \psi, \gamma, \rho, \delta)$$

$$\underline{entry}\,n\,\kappa\,(o, \ell : \phi, \psi, \gamma, \rho, \delta) = \#\phi \geq n \longrightarrow \kappa\,(o, \phi_a +\!\!+ \phi_r, \psi, \gamma, \rho, \delta),\ (o, \ell : \phi, \psi, \gamma, \rho, \delta)$$
$$\text{Where } \phi_a = map\,(Arg\,\gamma)\,(take\,n\,\phi),\quad \phi_r = drop\,(n-1)\,\phi$$

$$\underline{unwind}\,(o, \ell : \phi, \psi, \gamma, \rho, \delta) = (\nu \in \mathrm{Ide}) \longrightarrow \rho\,(\nu \mid \mathrm{Ide})\,(o, \ell : \phi, \psi, \gamma, \rho, \delta),$$
$$(\nu \in \mathbf{I}) \longrightarrow \underline{unwind}\,(o, (\nu \mid \mathbf{I}) : \phi, \psi, \gamma, \rho, \delta),$$
$$(\nu \in \mathbf{A}) \longrightarrow \underline{unwind}\,(o, fst\,(\nu \mid \mathbf{A}) : \phi, \psi, \gamma, \rho, \delta),\ (o, \ell : \phi, \psi, \gamma, \rho, \delta)$$
$$\text{Where } \nu = \gamma\,\ell$$

$$\underline{print}\,(o, \ell : \phi, \psi, \gamma, \rho, \delta) = (\nu \in \mathbf{Z}) \longrightarrow (o +\!\!+ (\nu \mid \mathbf{Z}), \phi, \psi, \gamma, \rho, \delta),$$
$$(\nu \in \mathbf{T} \vee \nu = nil) \longrightarrow (o, \phi, \psi, \gamma, \rho, \delta),$$
$$(\nu \in \mathbf{C}) \longrightarrow \kappa\,(o, \ell' : \ell'' : \phi, \psi, \gamma, \rho, \delta)), \stackrel{?}{=}$$
$$\text{Where } \nu = \gamma\,\ell,\quad \kappa = \underline{print} \circ \underline{eval} \circ \underline{print} \circ \underline{eval},\quad (\ell', \ell'') = (\nu \mid \mathbf{C})$$

Figure 9: Basic Instructions for Stack Semantics

of Figure 9 are clearly based on the compiler and abstract machine of Johnsson [1]. Perhaps the most important difference is the use of indirection nodes to implement the *update* instruction. The correctness of this alternative compilation technique is shown in Section 2. Other minor changes include

1. We have dispensed with a code component within the state, and instead used continuations. This includes an *if* instruction with two alternative continuations, rather than the use of labels as Johnsson does in [1].

2. The *entry* instruction incorporates some of the function assigned to *unwind* by Johnsson.

3. The local environment, β, encodes directly for the stack offset of a variable, because we

$New(\gamma)$ satisfies $\gamma(New(\gamma)) = \bot \wedge New(\lambda\ell.\bot) = \ell_0 \in \mathbf{L}$

$$Arg\,\gamma\,\ell = (\nu \in \mathbf{A}) \longrightarrow snd(\nu \mid \mathbf{A}), \stackrel{?}{=} \quad \text{Where } \nu = \gamma\,\ell$$

$$Elide\,\gamma\,\ell = (\nu \in \mathbf{I}) \longrightarrow Elide\,\gamma\,(\nu \mid \mathbf{I}), \ell \quad \text{Where } \nu = \gamma\,\ell$$

Figure 10: Auxiliary Definitions for Instructions

$$\begin{aligned}
\rho_{\text{init}}\,[\![\text{if}]\!] &= \underline{entry}\,3(\underline{exit}\,3 \circ \underline{if}(\underline{push}\,1)\,(\underline{push}\,2) \circ \underline{eval} \circ \underline{push}\,0)\\
\rho_{\text{init}}\,[\![\text{add}]\!] &= \underline{entry}\,2(\underline{exit}\,2 \circ \underline{add} \circ \underline{eval} \circ \underline{push}\,2 \circ \underline{eval} \circ \underline{push}\,0)\\
\rho_{\text{init}}\,[\![\text{eq}]\!] &= \underline{entry}\,2(\underline{exit}\,2 \circ \underline{eq} \circ \underline{eval} \circ \underline{push}\,2 \circ \underline{eval} \circ \underline{push}\,0)\\
\rho_{\text{init}}\,[\![\text{hd}]\!] &= \underline{entry}\,1(\underline{exit}\,1 \circ \underline{hd} \circ \underline{eval} \circ \underline{push}\,0)\\
\rho_{\text{init}}\,[\![\text{null}]\!] &= \underline{entry}\,1(\underline{exit}\,1 \circ \underline{null} \circ \underline{eval} \circ \underline{push}\,0)\\
\rho_{\text{init}}\,[\![\text{cons}]\!] &= \underline{entry}\,2(\underline{exit}\,2 \circ \underline{cons} \circ \underline{push}\,1 \circ \underline{push}\,1)
\end{aligned}$$

Figure 11: The Initial Environment ρ_{init}

$$\underline{if}\,\kappa_T\,\kappa_F\,(o, \ell : \phi, \psi, \gamma, \rho, \delta) = \nu \in \mathbf{T} \longrightarrow (\nu \mid \mathbf{T} \longrightarrow \kappa_T, \kappa_F)\,(o, \phi, \psi, \gamma, \rho, \delta), \stackrel{?}{=} \quad \text{Where } \nu = \gamma\,(Elide\,\gamma\,\ell)$$

$$\begin{aligned}
\underline{add}\,(o, \ell_0 : \ell_1 : \phi, \psi, \gamma, \rho, \delta) = {} & (\nu_0 \in \mathbf{Z} \wedge \nu_1 \in \mathbf{Z}) \longrightarrow \underline{pushvalue}\,\nu\,(o, \phi, \psi, \gamma, \rho, \delta), \stackrel{?}{=}\\
& \text{Where } \nu = (\nu_0 \mid \mathbf{Z} + \nu_1 \mid \mathbf{Z})\ \mathbf{in}\ \mathbf{N}\\
& \quad\ \nu_i = \gamma\,(Elide\,\gamma\,\ell_i)
\end{aligned}$$

$$\begin{aligned}
\underline{eq}\,(o, \ell_0 : \ell_1 : \phi, \psi, \gamma, \rho, \delta) = {} & (\nu_0 \in \mathbf{Z} \wedge \nu_1 \in \mathbf{Z}) \longrightarrow \underline{pushvalue}\,\nu\,(o, \phi, \psi, \gamma, \rho, \delta), \stackrel{?}{=}\\
& \text{Where } \nu = (\nu_0 \mid \mathbf{Z} = \nu_1 \mid \mathbf{Z})\ \mathbf{in}\ \mathbf{N}\\
& \quad\ \nu_i = \gamma\,(Elide\,\gamma\,\ell_i)
\end{aligned}$$

$$\underline{hd}\,(o, \ell : \phi, \psi, \gamma, \rho, \delta) = \nu \in \mathbf{C} \longrightarrow (o, fst\,(\nu \mid \mathbf{C}) : \phi, \psi, \gamma, \rho, \delta), \stackrel{?}{=} \quad \text{Where } \nu = \gamma\,(Elide\,\gamma\,\ell)$$

$$\underline{null}\,(o, \ell : \phi, \psi, \gamma, \rho, \delta) = \underline{pushvalue}\,(\nu = nil)\,(o, \phi, \psi, \gamma, \rho, \delta) \quad \text{Where } \nu = \gamma\,(Elide\,\gamma\,\ell)$$

$$\underline{cons}\,(o, \ell' : \ell'' : \phi, \psi, \gamma, \rho, \delta) = \underline{pushvalue}\,((\ell', \ell'')\ \mathbf{in}\ \mathbf{N})\,(o, \phi, \psi, \gamma, \rho, \delta)$$

Figure 12: Instructions for Built-in Functions

arrange for the offset to be incremented within β.

4. We have decoupled the dual role played by the dump in Johnsson's earlier work [2] as he himself does in [1]. This requires the introduction of a new stack **V** which is not used until Section 3.

To provide the implementation with primitive functions is straightforward. We define an initial environment ρ_{init}, for some representative functions, in Figure 11. The new instructions required are defined in Figure 12. The only point of interest is that the $\underline{eq}$ instruction is defined on integers only. In Section 2 we show how these functions can sometimes be coded in line.

One final point; the specification of *New* in Figure 10 is not as general as it might be. This is because we may be able to perform garbage collection, in which case *New* can return certain values not satisfying the definition given in Figure 10. The function *Arg* returns the argument component of an application node, whilst *Elide* removes any indirection nodes.

At this stage the question naturally arises; to what extent are the two specifications equivalent? The equivalence (or congruence) is proved in [3], and we state and sketch a proof now. The proof follows the pattern of congruence proofs established by [4], [8] and [9].

Theorem 1 *For all* Π *in* Prog

$$\acute{P}[\![\Pi]\!] = fst(\grave{P}[\![\Pi]\!])$$

Sketch of Proof We first note that because of its implementation by graph rewriting that we necessarily have $\acute{P}[\![\Pi]\!] \sqsupseteq fst(\grave{P}[\![\Pi]\!])$ by fixpoint (or computational) induction.

Although we could derive the stronger result $\acute{P}[\![\Pi]\!] \neq \bot \Rightarrow \acute{P}[\![\Pi]\!] \sqsupseteq fst(\grave{P}[\![\Pi]\!])$ this is still insufficiently strong, because an applicative order implementation would also satisfy the above condition.

The reverse inequality $\acute{P}[\![\Pi]\!] \sqsubseteq fst(\grave{P}[\![\Pi]\!])$ can be demonstrated by the use of the inclusive predicates c (to compare constructor nodes), e (to compare expressions) and f (to compare functions, by considering their application to congruent arguments). The existence of such predicates is then proved by induction over the complexity of the domains used in the direct semantics. Details of this induction are provided in [3]. □

As an aside, there are a number of ways to view the stack semantics. Firstly it may be considered to be a denotational description, specifying mathematical objects to which parts of the abstract syntax are mapped. Secondly, the instructions may be implemented as function definitions in a functional language and then we have an interpreter for the language. Or thirdly, the code may be written out as a string and an abstract machine (specified by the auxiliary definitions of Figures 9 and 10) executes these sequences. This is possible because the state is used completely strictly, in the sense that each intermediate state may be completely evaluated.

2 Partial Compile-time Reduction

In this section we analyse the correctness of some of the simple code improvements suggested by Johnsson [1] and Peyton Jones [6].

Before we attempt to do this we must establish the form of such an equivalence. Two graphs are *isomorphic* if they are the same except for the names we give to the labels for the nodes. In the

current situation we must extend this idea to cover indirection nodes and garbage or unreachable parts of the graph. We first define a function to remove indirection nodes from a state.

Definition 1 We define *ri* as follows

$$\begin{aligned} ri(o, \phi, \psi, \gamma, \rho, \delta) &= (o, \phi', \psi, \gamma', \rho, \delta') \\ &\quad \text{Where } \phi' = map\,(Elide\,\gamma)\,\phi \\ &\quad\quad \gamma' = \lambda\ell.(\gamma\,\ell \in \mathbf{I} \longrightarrow Elide\,\gamma(\gamma\,\ell),\ \gamma\,\ell) \\ &\quad\quad \delta' = map\,(map\,(Elide\,\gamma))\,\phi \end{aligned}$$

We now define the set of reachable nodes for a state, using *mark*.

Definition 2 For a given stack ϕ in $\mathbf{L}^*$, dump δ in $\mathbf{D}$, and graph map γ in $\mathbf{G}$, the set $mark(\gamma, \phi : \delta)$, is defined as follows:

1. If ℓ is a label contained in $\phi : \delta$, then ℓ is in the set $mark(\gamma, \phi : \delta)$.
2. If ℓ is in $mark(\gamma, \phi : \delta)$, then let $\nu = \gamma\,\ell$.
   (a) If $\nu \in \mathbf{A}$, then ℓ' and ℓ'' are in $mark(\gamma, \phi : \delta)$, where $(\ell', \ell'') = (\nu \mid \mathbf{A})$.
   (b) If $\nu \in \mathbf{I}$, then ℓ is in $mark(\gamma, \phi : \delta)$, where $\ell = (\nu \mid \mathbf{I})$.
   (c) If $\nu \in \mathbf{C}$, then ℓ' and ℓ'' are in $mark(\gamma, \phi : \delta)$, where $(\ell', \ell'') = (\nu \mid \mathbf{C})$.
3. No other labels are in the set $mark(\gamma, \phi : \delta)$.

We now define a function to assign $\perp$ to all non-reachable nodes of the graph.

Definition 3 We define *gc* as follows

$$\begin{aligned} gc(o, \phi, \psi, \gamma, \rho, \delta) &= (o, \phi, \psi, \gamma', \rho, \delta) \\ &\quad \text{Where } \gamma' = \lambda\ell.(\ell \in mark(\gamma, \phi : \delta) \longrightarrow \gamma\,\ell,\ \perp) \end{aligned}$$

Two states, σ_0 and σ_1 are then said to be *graph-isomorphic* if and only if there exists a relabelling of the nodes in σ_1' such that this is identical to σ_0', where $\sigma_i' = gc(ri(\sigma_i))$. We denote this by writing $\sigma_0 \cong \sigma_1$. Extending this notation, we may say that two continuations, κ and κ' from $\mathbf{K}$ are graph-isomorphic if, for all σ, $\kappa\,\sigma \cong \kappa'\,\sigma$.

We now state some continuation pairs that are graph-isomorphic.

Theorem 2 *For all* E, E_0, E_1 *and* E_2 *in* Exp, *all* β *in* **B** *and all* $n > 0$

$$\begin{aligned} \mathcal{E}\,[\![\mathrm{if}\,\mathrm{E}_0\,\mathrm{E}_1\,\mathrm{E}_2]\!]\,\beta &\cong \underline{if}(\mathcal{E}\,[\![\mathrm{E}_1]\!]\,\beta)\,(\mathcal{E}\,[\![\mathrm{E}_2]\!]\,\beta) \circ \mathcal{E}\,[\![\mathrm{E}_0]\!]\,\beta \\ \mathcal{E}\,[\![\mathrm{add}\,\mathrm{E}_0\,\mathrm{E}_1]\!]\,\beta &\cong \underline{add} \circ \mathcal{E}\,[\![\mathrm{E}_0]\!]\,(inc \circ \beta) \circ \mathcal{E}\,[\![\mathrm{E}_1]\!]\,\beta \\ \mathcal{E}\,[\![\mathrm{eq}\,\mathrm{E}_0\,\mathrm{E}_1]\!]\,\beta &\cong \underline{eq} \circ \mathcal{E}\,[\![\mathrm{E}_0]\!]\,(inc \circ \beta) \circ \mathcal{E}\,[\![\mathrm{E}_1]\!]\,\beta \\ \mathcal{E}\,[\![\mathrm{hd}\,\mathrm{E}]\!]\,\beta &\cong \underline{eval} \circ \underline{hd} \circ \mathcal{E}\,[\![\mathrm{E}]\!]\,\beta \\ \mathcal{E}\,[\![\mathrm{null}\,\mathrm{E}]\!]\,\beta &\cong \underline{null} \circ \mathcal{E}\,[\![\mathrm{E}]\!]\,\beta \\ \mathcal{E}\,[\![\mathrm{cons}\,\mathrm{E}_0\,\mathrm{E}_1]\!]\,\beta &\cong \underline{cons} \circ \mathcal{C}\,[\![\mathrm{E}_0]\!]\,(inc \circ \beta) \circ \mathcal{C}\,[\![\mathrm{E}_1]\!]\,\beta \\ \mathcal{R}\,[\![\mathrm{if}\,\mathrm{E}_0\,\mathrm{E}_1\,\mathrm{E}_2]\!]\,\beta\,n &\cong \underline{if}(\mathcal{R}\,[\![\mathrm{E}_1]\!]\,\beta\,n)\,(\mathcal{R}\,[\![\mathrm{E}_2]\!]\,\beta\,n)) \circ \mathcal{E}\,[\![\mathrm{E}_0]\!]\,\beta \end{aligned}$$

Proof All the proofs follow directly from the definitions of the instructions, and graph-isomorphism, there being no induction necessary.

As an example we consider the first pair. Letting $\beta^{+n} = (\lambda x.x + n) \circ \beta$, we have:

$$\mathcal{E}\,[\![\text{if } \mathrm{E}_0\,\mathrm{E}_1\,\mathrm{E}_2]\!]\,\beta = \underline{\textit{eval}} \circ \underline{\textit{mkap}} \circ \underline{\textit{mkap}} \circ \underline{\textit{mkap}} \circ \underline{\textit{pushvalue}}\,(\text{if in N}) \circ \mathcal{C}\,[\![\mathrm{E}_0]\!]\,\beta^{+2} \circ \mathcal{C}\,[\![\mathrm{E}_1]\!]\,\beta^{+1} \circ \mathcal{C}\,[\![\mathrm{E}_2]\!]\,\beta$$

Let us assume that just prior to the execution of *eval* the state is of the form $(o, \ell : \phi, \psi, \gamma', \rho, \delta)$, such that the roots of the graphs for expressions E_i are ℓ_i. Then after executing *unwind* and *entry* we have

$$\underline{\textit{restore}} \circ \kappa\,(o, [\ell_0, \ell_1, \ell_2, \ell], \psi, \gamma', \rho, \phi : \delta)$$

where κ is $\underline{\textit{exit}}3 \circ \underline{\textit{if}}(\underline{\textit{push}}1)\,(\underline{\textit{push}}2) \circ \underline{\textit{eval}} \circ (\underline{\textit{push}}0)$. We then consider, by cases, the result of executing the first two instructions of this sequence. If the result is a proper value and a member of **T**, we then insert an indirection node from ℓ to ℓ_i $i = 1, 2$. Suppose the result on top of the stack were true. This gives

$$\underline{\textit{restore}} \circ \underline{\textit{unwind}}(o, [\ell], \psi, \gamma'' \oplus \{\ell \mapsto \ell_1 \text{ in N}\}, \rho, \phi : \delta)$$

but by the definition of *eval* this is graph-isomorphic to

$$\underline{\textit{eval}}(o, \ell_1 : \phi, \psi, \gamma'' \oplus \{\ell \mapsto \ell_1 \text{ in N}\}, \rho, \delta),$$

which in turn is graph-isomorphic to the right hand side of the continuation pair, under the assumption that ℓ_0 reduces to true. By similar arguments this result applies when ℓ_0 reduces to false, $\underline{?}$ or $\perp$. □

We now study a generalisation (and formalisation) of an observation made in [11]. They observe that indirection nodes accumulate rapidly during SKI combinator reduction, and propose as a solution that the first argument to the K combinator be reduced before reduction of $K\,x\,y$ is attempted. One could then copy the root of the answer rather than insert an indirection node. This was taken up independently by Johnsson and incorporated into the G-machine design. Our semantics from Figure 8 makes an alternative design decision, for two reasons. Firstly the resulting machine is semantically cleaner, leading directly to a congruence proof. Secondly, it may execute faster for higher-order functions, a point to which we will return. We now prove Theorem 3, which demonstrates that we may reduce the body of a combinator before overwriting the root. As the resulting piece of graph will then be in head normal form we may use a copying form of *update* without losing sharing. This I have called Johnsson's technique.

Theorem 3 *For all* E *in* Exp, *all* β *in* **B**, *and all* $m > 0$

$$\mathcal{R}\,[\![\mathrm{E}]\!]\,\beta\,m \cong \underline{\textit{exit}}\,m \circ \mathcal{E}\,[\![\mathrm{E}]\!]\,\beta$$

Proof We first observe that this is equivalent to showing $\underline{exit}\, m \cong \underline{exit}\, m \circ \underline{eval}$ by expanding $\mathcal{R}$ and $\mathcal{E}$. Suppose that the state to which we apply each continuation is of the form

$$(o, \ell : \phi +\!\!+ \phi', \psi, \gamma, \rho, \delta),$$

with $\#(\phi) = m$.

We are required to show that

$$\underline{unwind}\,\sigma^0_{\mathcal{R}} \cong (\underline{exit}\, m \circ \underline{restore})\,\sigma^0_{\mathcal{E}}$$

where

$$\begin{aligned} \sigma^0_{\mathcal{R}} &= (o, \ell : \phi', \psi, \gamma^0_{\mathcal{R}}, \rho, \delta) \quad \text{and} \\ \sigma^0_{\mathcal{E}} &= (o, [\ell], \psi, \gamma^0_{\mathcal{E}}, \rho, (\phi +\!\!+ \phi') : \delta) \end{aligned}$$

with $\gamma^n_{\mathcal{R}} = \gamma^n_{\mathcal{E}} \oplus \{(hd\phi') \mapsto \ell \textbf{ in } \mathbf{N}\}$.

We now observe that $\underline{unwind}\sigma^0_{\mathcal{R}}$ and $\underline{unwind}\sigma^0_{\mathcal{E}}$ execute in the same way producing states of the form:

$$\begin{aligned} \sigma^n_{\mathcal{R}} &= (o, \phi_n \ell : \phi', \psi, \gamma^n_{\mathcal{R}}, \rho, \delta) \\ \sigma^n_{\mathcal{E}} &= (o, \phi_n +\!\!+ [\ell], \psi, \gamma^n_{\mathcal{E}}, \rho, (\phi +\!\!+ \phi') : \delta) \end{aligned}$$

or

$$\begin{aligned} \sigma^n_{\mathcal{R}} &= (o, \phi_n, \psi, \gamma^n_{\mathcal{R}}, \rho, \delta_n +\!\!+ [\phi'_n +\!\!+ [\ell] +\!\!+ \phi'] +\!\!+ \delta) \\ \sigma^n_{\mathcal{E}} &= (o, \phi_n, \psi, \gamma^n_{\mathcal{E}}, \rho, \delta_n +\!\!+ [\phi'_n +\!\!+ [\ell], \phi +\!\!+ \phi'] +\!\!+ \delta) \end{aligned}$$

until one of the following conditions occurs.

1. The evaluation has completed, in which case

   $$\underline{unwind}\,\sigma^n_{\mathcal{E}} = \sigma^n_{\mathcal{E}}$$

   and

   $$\sigma^n_{\mathcal{E}} = (o, \phi_n +\!\!+ [\ell], \psi, \gamma^n_{\mathcal{E}}, \rho, (\phi +\!\!+ \phi') : \delta)$$

   If this is so

   $$\sigma^n_{\mathcal{R}} = (o, \phi_n, \psi, \gamma^n_{\mathcal{R}}, \rho, \delta_n +\!\!+ [\phi'_n +\!\!+ [\ell] +\!\!+ \phi'] +\!\!+ \delta)$$

   and executing $\underline{pop}\, m \circ \underline{update}\, m \circ \underline{restore}$ on $\sigma^n_{\mathcal{E}}$ gives a state graph isomorphic to $\sigma^n_{\mathcal{R}}$.

2. We reach states $\sigma^n_{\mathcal{R}}$ and $\sigma^n_{\mathcal{E}}$ with $hd\,\phi_n = hd\,\phi'$ to which we apply *unwind*. The execution of the two forms then diverges, because $\gamma^n_{\mathcal{R}}\,(hd\,\phi') \neq \gamma^n_{\mathcal{E}}\,(hd\,\phi')$.

   Fortunately this loss of sharing is unimportant because it only occurs in programs that fail to terminate.

   To see this recall that we are attempting to reduce $(hd\,\phi')$. If, during this reduction, we attempt to reduce it again then we have an infinite regress.

□

Notice the inefficiency that the $\mathcal{E}$-scheme version has when dealing with the return of higher-order functions. First the stack is cleared of all except the last element. Next we restore the previous stack, performing some adjustments to it. Finally, we proceed to reconstruct the spine we have just thrown away. Contrast this with the way that the $\mathcal{R}$-scheme proceeds. We have to state, on the other hand, that the use of indirection nodes gives rise to the possibility of chains of indirection nodes building up, and the cost of accessing arguments to a function is therefore increased.

If the result is not a function then the $\mathcal{E}$-scheme will work best, especially if we have a special *unwind* (called *ret* in [6]) that doesn't check that the result is already unwound.

At this stage we have seen that the built-in functions can sometimes be compiled *in-line* or perhaps more descriptively, that there exist correct partial compile-time reductions.

3 A Stack for Basic Values

So far we have not used the third component of the state at all, this is now remedied. It is intended that this component should be a stack of integer or boolean values, and so we now provided instructions and alternative compilations for this use.

Before investigating graph-isomorphic continuations, we digress a little to mention typing. We will presume that our language is polymorphically typed in the manner described by Milner in [5]. We shall presume that we have a function $\mathcal{T}$ that returns the type of an expression. For most compilation purposes we are only interested in these types if they are Int or Bool.

Definition 4 If $\mathcal{T}\,[\![\mathrm{E_I}]\!] = \mathrm{Int}$ and $\mathcal{T}\,[\![\mathrm{E_B}]\!] = \mathrm{Bool}$, then for all $\mathrm{E_I}$, all $\mathrm{E_B}$ and all β

$$\begin{array}{lcl}
\mathcal{B}\,[\![\mathrm{E_I}]\!]\,\beta & \cong & \underline{get} \circ \mathcal{E}\,[\![\mathrm{E_I}]\!]\,\beta \\
\mathcal{B}\,[\![\mathrm{E_B}]\!]\,\beta & \cong & \underline{get} \circ \mathcal{E}\,[\![\mathrm{E_B}]\!]\,\beta \\
\underline{mkint} \circ \mathcal{B}\,[\![\mathrm{E_I}]\!]\,\beta & \cong & \mathcal{E}\,[\![\mathrm{E_I}]\!]\,\beta \\
\underline{mkbool} \circ \mathcal{B}\,[\![\mathrm{E_B}]\!]\,\beta & \cong & \mathcal{E}\,[\![\mathrm{E_B}]\!]\,\beta
\end{array}$$

We may now state Theorem 3.

$$\underline{ifv}\,\kappa_T\,\kappa_F\,(o,\phi,\tau:\psi,\gamma,\rho,\delta) = \tau \in \mathbf{T} \longrightarrow (\tau \mid \mathbf{T} \longrightarrow \kappa_T, \kappa_F)\,\sigma, \underline{?} \quad \text{Where } \sigma = (o,\phi,\psi,\gamma,\rho,\delta)$$

$$\underline{addv}\,(o,\phi,\varsigma_0:\varsigma_1:\psi,\gamma,\rho,\delta) = (\varsigma_0 \in \mathbf{Z} \wedge \varsigma_1 \in \mathbf{Z}) \longrightarrow (o,\phi,\varsigma:\psi,\gamma,\rho,\delta), \underline{?} \quad \text{Where } \varsigma = \varsigma_0 \mid \mathbf{Z} + \varsigma_1 \mid \mathbf{Z}$$

$$\underline{eqv}\,(o,\phi,\varsigma_0:\varsigma_1:\psi,\gamma,\rho,\delta) = (\varsigma_0 \in \mathbf{Z} \wedge \varsigma_1 \in \mathbf{Z}) \longrightarrow (o,\phi,\tau:\psi,\gamma,\rho,\delta), \underline{?} \quad \text{Where } \tau = \varsigma_0 \mid \mathbf{Z} = \varsigma_1 \mid \mathbf{Z}$$

$$\underline{nullv}\,(o,\ell:\phi,\psi,\gamma,\rho,\delta) = (o,\phi,(\nu = \mathit{nil}):\psi,\gamma,\rho,\delta) \quad \text{Where } \nu = \gamma\,(\mathit{Argi}\,\gamma\,\ell_i)$$

$$\underline{mkint}\,(o,\phi,\varsigma:\psi,\gamma,\rho,\delta) = (\varsigma \in \mathbf{Z}) \longrightarrow \underline{pushvalue}\,(\varsigma\ \mathbf{in}\ \mathbf{N})\,(o,\phi,\psi,\gamma,\rho,\delta), \underline{?}$$

$$\underline{mkbool}\,(o,\phi,\tau:\psi,\gamma,\rho,\delta) = (\tau \in \mathbf{T}) \longrightarrow \underline{pushvalue}\,(\tau\ \mathbf{in}\ \mathbf{N})\,(o,\phi,\psi,\gamma,\rho,\delta), \underline{?}$$

$$\underline{get}\,(o,\ell:\phi,\psi,\gamma,\rho,\delta) = \begin{array}{l}(\nu \in \mathbf{Z}) \longrightarrow (o,\phi,\nu \mid \mathbf{Z}:\psi,\gamma,\rho,\delta) \\ (\nu \in \mathbf{T}) \longrightarrow (o,\phi,\nu \mid \mathbf{T}:\psi,\gamma,\rho,\delta), \underline{?} \\ \quad \text{Where } \nu = \gamma\,(\mathit{Argi}\,\gamma\,\ell)\end{array}$$

Figure 13: Instructions for Built-in Functions (using **V**)

Theorem 4 *For all well-typed programs, the following pairs of continuations are equivalent:*

$$\begin{array}{lcl}
\mathcal{B}\,[\![\mathrm{add}\,\mathrm{E}_0\,\mathrm{E}_1]\!]\,\beta & \cong & \underline{addv} \circ \mathcal{B}\,[\![\mathrm{E}_0]\!]\,\beta \circ \mathcal{B}\,[\![\mathrm{E}_1]\!]\,\beta \\
\mathcal{B}\,[\![\mathrm{eq}\,\mathrm{E}_0\,\mathrm{E}_1]\!]\,\beta & \cong & \underline{eqv} \circ \mathcal{B}\,[\![\mathrm{E}_0]\!]\,\beta \circ \mathcal{B}\,[\![\mathrm{E}_1]\!]\,\beta \\
\mathcal{R}\,[\![\mathrm{if}\,\mathrm{E}_0\,\mathrm{E}_1\,\mathrm{E}_2]\!]\,\beta\,n & \cong & \underline{ifv}\,(\mathcal{R}\,[\![\mathrm{E}_1]\!]\,\beta\,n)\,(\mathcal{R}\,[\![\mathrm{E}_2]\!]\,\beta\,n) \circ \mathcal{B}\,[\![\mathrm{E}_0]\!]\,\beta \\
\mathcal{E}\,[\![\mathrm{if}\,\mathrm{E}_0\,\mathrm{E}_1\,\mathrm{E}_2]\!]\,\beta & \cong & \underline{ifv}\,(\mathcal{E}\,[\![\mathrm{E}_1]\!]\,\beta)\,(\mathcal{E}\,[\![\mathrm{E}_2]\!]\,\beta) \circ \mathcal{B}\,[\![\mathrm{E}_0]\!]\,\beta \\
\mathcal{B}\,[\![\mathrm{if}\,\mathrm{E}_0\,\mathrm{E}_1\,\mathrm{E}_2]\!]\,\beta & \cong & \underline{ifv}\,(\mathcal{B}\,[\![\mathrm{E}_1]\!]\,\beta)\,(\mathcal{B}\,[\![\mathrm{E}_2]\!]\,\beta) \circ \mathcal{B}\,[\![\mathrm{E}_0]\!]\,\beta \\
\mathcal{B}\,[\![\mathrm{null}\,\mathrm{E}]\!]\,\beta & \cong & \underline{nullv} \circ \mathcal{E}\,[\![\mathrm{E}]\!]\,\beta
\end{array}$$

Proof All these are demonstrated using the definitions already given. □

This concludes our brief survey of some of the code improvement techniques used in the G-machine.

4 Conclusion

We have shown that graph reduction can be represented using a stack semantics. The form of the specification suggests a compiler and abstract machine. Alternative semantics, *e.g.* for abstract interpretation, could also be built around the model provided here.

As a consequence of this representation, we were able to show that some of Johnsson's code improvements were correct. The notion of graph isomorphism leads to a concise framework for reasoning about such code improvements. It is interesting to note that the technical difficulties with one of these improvements resulted from a loss of sharing which occurred only for certain non-terminating programs. Using this framework it is possible to prove Johnsson and Peyton

Jones' tail recursion improvements correct, although it is not done in this paper.

Obviously these results can be extended in a number of directions. These include extending the language, providing more code improvements and increasing the level of detail contained in the instruction definitions. Another avenue of pursuit would be to obtain a similar derivation for a parallel machine, using multiple stacks.

Acknowledgements

I would like to thank Phil Wadler, Simon Peyton-Jones and three anonymous reviewers for their helpful comments on draft forms of this paper.

References

[1] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, January 1987.

[2] Thomas Johnsson. The G-machine. In *Proceedings of the Workshop on Declarative Programming*, University College, London, April 1983.

[3] David Lester. A congruence proof for graph reduction. 1987. Unpublished.

[4] R.E. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, 1976.

[5] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[6] Simon Peyton Jones. *The Implementation of Functional Languages.* Prentice Hall, Englewood Cliffs, NJ, USA, 1987.

[7] Simon L. Peyton Jones. Directions in functional programming research. In David A. Duce, editor, *Distributed Computing Systems Programme*, chapter 14, pages 220–249, Peter Peregrinus Ltd., London, UK., 1984.

[8] Joseph E. Stoy. The congruence of two programming language definitions. No. 4343.

[9] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. The MIT Series in Computer Science*, The MIT Press, Cambridge, Massachusetts, 1977.

[10] Joseph E. Stoy. *Some Mathematical Aspects of Functional Programming.* Lecture Notes, University of Newcastle upon Tyne, July 1981.

[11] W.R. Stoye, T.J.W. Clarke, and A.C. Norman. Some practical methods for rapid combinator reduction. In *Proceedings of the A.C.M. Symposium on Lisp and Functional Languages*, Austin, August 1984.