CORRECTNESS OF CODE GENERATION
FROM A TWO-LEVEL META-LANGUAGE

(Extended Abstract)

Flemming Nielson

SUMMARY

We present a two-level denotational metalanguage suitable for defining programming languages with procedures as second class citizens. Code generation is developed for a slight restriction of this metalanguage and this generalizes previous algebraic approaches. This paper focuses on the correctness proof of the code generation; in particular the method used for directly proving equivalence between an operational and a denotational semantics and the modification of Reynolds' idea of a relational functor.

INTRODUCTION

Many papers have considered the use of an "algebraic framework" for expressing compiler construction and in particular for structuring the proof of correctness. An early paper is (Morris) which considers compilation of an iterative language into flow charts and the use of initiality and homomorphisms for proving the correctness. The algebraic formulation and proof of correctness was improved by Thatcher et al. The link to denotational semantics became apparent in (Mosses), and also Dybjer uses ideas from denotational semantics (in particular from domain theory). And there have been many other papers.

These papers just consider simple imperative languages. Hence there is a long way to go before more realistic languages can be handled systematically. (The impressive development of Milne and Strachey is too ad hoc to be called systematic.) This gap is bridged somewhat in this paper. It is done by considering a certain meta-language that has been found useful for a different purpose (Nielson, 1984). For a slight restriction of this meta-language code generation can be developed and this paper reports on the correctness proof. The restricted meta-language is applicable to a wide class of languages, including a variant of SMALL (Gordon); for details see the paper by Nielson and Nielson in these proceedings.

THE META-LANGUAGE

The untraditional aspect of the meta-language is the types. Usually a denotational meta-language is a typed λ -calculus with types given by for example

$$t ::= A_i \mid t+t' \mid t \times t' \mid t \rightarrow t' \mid \text{rec } X.t \mid X$$

In the present meta-language there are two kinds of types: "compile-time" types ct and "run-time" types rt . The idea is that rt and ct are "copies" of t and that run-time entities can be considered at compile-time but not vice versa. This leads to

$$ct ::= A_i \mid ct+ct' \mid ct \times ct' \mid ct \rightarrow ct' \mid \text{rec } X.ct \mid X \mid rt$$

$$rt ::= \underline{A_i} \mid rt+\underline{rt'} \mid rt \times \underline{rt'} \mid rt \rightarrow \underline{rt'} \mid \underline{\text{rec } X}.rt \mid \underline{X}$$

and then e.g. the expression procedures of (Tennent) may be given type $S \times E \rightarrow \underline{S \times E}$ whereas static expressions procedures are given type $S \times E \rightarrow S \times E$.

The idea of two kinds of types has proved very useful in (Nielson, 1984) for developing abstract interpretation (Cousot et al.) for denotational definitions. Technical problems with $rt \rightarrow rt'$ motivated using the weaker system TMLs defined by

$$ct ::= A_i \mid ct+ct' \mid ct \times ct' \mid ct \rightarrow ct' \mid \text{rec } X.ct \mid X \mid \underline{rt \rightarrow rt'}$$

$$rt ::= \underline{A_i} \mid rt+\underline{rt'} \mid rt \times \underline{rt'} \mid \underline{\text{rec } X}.rt \mid \underline{X}$$

and this system would seem to be applicable to all sequential languages that use procedures in a "second-class fashion" (e.g. PASCAL). We shall take this type system as our starting point.

The expressions of TMLs intuitively fall into two parts. One part corresponds to ct and is an ordinary typed λ -calculus. The other part corresponds to $rt \rightarrow rt'$ and is close to FP (Backus). The syntax is

$$\begin{aligned} e ::= & f_i \mid (e, e') \mid e \downarrow i \mid in_i e \mid is_i e \mid out_i e \mid \lambda x. e \mid e(e') \mid x \mid \\ & mkrec e \mid unrec e \mid e \rightarrow e', e'' \mid fix_{ct} e \\ & \mid tuple(e, e') \mid take_i \mid in_i \mid case(e, e') \mid mkrec \mid unrec \mid \\ & cond(e, e', e'') \mid e \sqsupset e' \end{aligned}$$

where we omit stating the type constraints (Nielson, 1984). Concerning the FP-like notation the intention (in the standard semantics) is that $tuple(e, e')(v) = (e(v), e'(v))$ and $take_1(v, v') = v$ etc.

Code generation and the correctness proof will be performed for a meta-language TMLsc that is like TMLs except that expressions may only use fix_{ct} for so-called composite types ct (see later). The restricted meta-language is sufficiently powerful that a variant of SMALL (Gordon) can be defined. We refer to (Nielson and Nielson, these proceedings) for an analysis of the "naturalness" of using these meta-languages.

SEMANTICS

We shall follow the approach of Reynolds and model the meaning of ct by a domain functor. This is a special kind of covariant functor over a category $\underline{CPO2s}$ with cpo 's as objects and pairs $(f: D \rightarrow D', f': D' \rightarrow D)$ of strict continuous functions as the morphisms from D to D' . So we define $\underline{I}[[ct]] : \underline{CPO2s}^N \rightarrow \underline{CPO2s}$ where N is the number of free domain variables and $\underline{I}$ interprets $rt \rightarrow rt'$. The equations are

$$\begin{aligned} \underline{I}[[A_i]] &= \text{a constant functor} \\ \underline{I}[[ct+ct']] &= +^S \circ (\underline{I}[[ct]], \underline{I}[[ct']]) \\ \underline{I}[[ct \times ct']] &= \times^S \circ (\underline{I}[[ct]], \underline{I}[[ct']]) \\ \underline{I}[[ct \rightarrow ct']] &= \rightarrow^S \circ (\underline{I}[[ct]], \underline{I}[[ct']]) \\ \underline{I}[[rec X.ct]] &= REC(\underline{I}[[ct]]) \\ \underline{I}[[x]] &= \text{corresponding projection functor} \\ \underline{I}[[rt \rightarrow rt']] &= \underline{I}(rt \rightarrow rt') \end{aligned}$$

Here $\circ$ denotes composition of functors and $(\dots, \dots)$ denotes tupling. Further, $\times^S(D, D') = D \times D'$ and $\times^S((f, f'), (g, g')) = (\lambda(u, u'). (f(u), f'(u')), \dots)$ and $\rightarrow^S(D, D') = D \rightarrow D'$ and $\rightarrow^S((f, f'), (g, g')) = (\lambda h. g \circ h \circ f', \dots)$. Finally, $REC(F)(D)$ is the limiting object of the chain $\{\cdot, F(D, \{\cdot\}), \dots$ and $REC(F)(f)$ is $\bigcup_n r_n' \circ (\lambda g. F(f, g))^n \cdot r_n^u$ where r_n' and r_n^u are appropriate embeddings and r_n^u an upper adjoint. We refer to Smyth and Plotkin (1982) for any domain theory not covered here.

In the interpretation $\underline{K}$ for code generation the idea is to define $\underline{K}(\underline{rt} \rightarrow \underline{rt}')$ to be a flat domain of code. In the standard interpretation $\underline{S}$ we shall let $\underline{S}(\underline{rt} \rightarrow \underline{rt}')$ be a domain of functions from $\underline{S}[[\underline{rt}]]$ to $\underline{S}[[\underline{rt}']]$. It may be partial functions between sets or strict continuous functions between domains. In both cases $\underline{S}[[\underline{rt}]] : \underline{B}^N \rightarrow \underline{B}$ is defined much like $\underline{S}[[\underline{ct}]]$ (except that in $\underline{B}$ there is no need for morphisms to be pairs of functions).

Consider next an expression e that has type ct whenever the free variables $x_1, \dots, x_n$ have types $ct_1, \dots, ct_n$. The semantic function

$$\underline{I}[[e]] : \underline{I}[[ct_1]] \times \dots \times \underline{I}[[ct_n]] \rightarrow \underline{I}[[ct]]$$

is defined structurally upon e . For expressions intuitively corresponding to ct the definition is mostly trivial, e.g.

$$\underline{I}[[(e, e')]] (env) = (\underline{I}[[e]] (env), \underline{I}[[e']] (env))$$

$$\underline{I}[[e \downarrow i]] (env) = (\underline{I}[[e]] (env)) \downarrow i$$

The semantics of the constants f_i are not the same in all interpretations so we write

$$\underline{I}[[f_i]] = \underline{I}(f_i)$$

to indicate that the interpretation must specify their effect. Similarly we write e.g.

$$\underline{I}[[\text{tuple}(e, e')]] (env) = \underline{I}(\text{tuple})(\underline{I}[[e]] (env), \underline{I}[[e']] (env))$$

$$\underline{I}[[\text{take}_i]] (env) = \underline{I}(\text{take}_i).$$

In the standard interpretation we have $\underline{S}(\text{tuple})(f, f')$ to be $\lambda v. (f(v), f'(v))$ and $\underline{S}(\text{take}_i)$ to be $\lambda v. v \downarrow i$. Hopefully, the definitions of $\underline{I}[[e]]$ and $\underline{S}$ have by now been sketched in sufficient detail.

CODE GENERATION

We now turn towards the code generation interpretation $\underline{K}$. The idea is that the domain $\underline{K}(\underline{rt} \rightarrow \underline{rt}')$ will be a flat domain of instruction sequences. The particular machine to be used here has the following instructions:

$$\begin{aligned} \text{ins} ::= & \text{enter} \mid \text{switch} \mid \text{tak}(i) \mid \text{tup}(k) \mid \text{push}(v) \mid \text{branch}(\ell_1, \dots, \ell_k) \\ & \mid \text{def}(\ell) \mid \text{goto}(\ell) \mid \text{branchfalse}(\ell) \mid \text{ope}(w) \mid \text{call}(\ell) \mid \text{rtn} \end{aligned}$$

A configuration of the machine is a triple (pc, st, cs) of program counter, stack and control-stack. Then "enter" duplicates the element on top of st , "switch" changes the order of the topmost two elements, "tak(i)" replaces the topmost element by its i 'th component etc. The instruction "call(ℓ)" places the return address $pc+1$ on top of cs and rtn pops cs and transfers control as indicated by the popped value.

Formally, the semantics of a program ϕ (i.e. an instruction sequence ϕ) is a rewrite relation $\rightarrow$ (or $\rightarrow_\phi$) upon configurations. We write $\nrightarrow$ when no next configuration

exists and $\gamma \rightarrow^{**} \gamma'$ when $\gamma \rightarrow^* \gamma' \nrightarrow$. Based on this we define

$$\text{RUN}(\phi, pc)(st, cs) = \begin{cases} \perp & \text{if there is no } \gamma \text{ s.t. } (pc, st, cs) \rightarrow^{**} \gamma \\ & \text{assume next } (pc, st, cs) \rightarrow^{**} (pc', st', cs') \text{ then} \\ (st', cs') & \text{if } pc' = 1 + \text{length}(\phi) \\ \text{error}_{lab} & \text{if } \phi \downarrow pc' \text{ transfers control to a label} \\ & \text{that has not been defined} \\ \text{error}_{cs} & \text{if } \phi \downarrow pc' = \text{rtn} \\ \text{error}_{st} & \text{if } \exists \overline{st}, \gamma: (pc', st', \overline{st}, cs') \rightarrow \gamma \\ \text{error}_r & \text{otherwise} \end{cases}$$

This function is extended to be the identity upon arguments $\perp$, error_{lab} , error_{cs} , error_{st} , and error_r . The idea is that for a "well-composed" program ϕ none of error_{lab} , error_{cs} , or error_{st} can occur.

Consider now the definition of the primitives $\underline{K}(\text{tuple})$ etc. To make this work we shall use $\underline{K}(\text{rt} \rightarrow \text{rt}') = \text{Occ} \rightarrow \text{Ins}^*$ where Ins^* is the flat domain of instruction sequences and Occ is a flat domain of occurrences (i.e. finite sequences of numbers and letters). Occurrences are used because there are no "global variables" in denotational semantics and we need some way of ensuring uniqueness of the labels generated. Some example primitives then are

$$\underline{K}(\text{tuple})(g, g') = \lambda \text{occ}. [\text{enter}]^{\wedge} (g' \text{occ}^{\wedge} [1])^{\wedge} [\text{switch}]^{\wedge} (g \text{occ}^{\wedge} [2])^{\wedge} [\text{tup}(2)]$$

$$\underline{K}(\text{take}_i) = \lambda \text{occ}. [\text{tak}(i)].$$

Except from fix there are no profound difficulties in completing the definition of $\underline{K}$. The details are covered in (Nielson and Nielson, 1985).

To handle fix_{ct} of type $(ct \rightarrow ct) \rightarrow ct$ we shall need the predicates pure (abbreviated p) and composite (abbreviated c):

ct	A_i	$ct' \times ct''$	$ct' + ct''$	$ct' \rightarrow ct''$	$\text{rec } X.ct'$	X	$\text{rt} \rightarrow \text{rt}'$
$p(ct)$	true	$p(ct') \wedge p(ct'')$	$(as \times)$	$(as \times)$	$p(ct')$	true	false
$c(ct)$	true	$c(ct') \wedge c(ct'')$	$(as \times)$	false	false	false	true

The idea is that ct is pure if there is no $\text{rt} \rightarrow \text{rt}'$ in it. Then define $\underline{K}(\text{fix}_{ct})(G)$ as follows:

- if ct is pure then $\text{LFP}(G)$
- if $ct = ct' \times ct''$ then $\langle G', G''(G') \rangle$
 - where $G''(g') = \underline{K}(\text{fix}_{ct''})(\lambda g''. G(g', g'') \downarrow 2)$
 - and $G' = \underline{K}(\text{fix}_{ct'})(\lambda g'. G(g', G''(g')) \downarrow 1)$
- if $ct = ct' + ct''$ then $\text{is}_1(GL) \rightarrow \text{in}_1(\underline{K}(\text{fix}_{ct'})(\text{out}_1 \cdot G \cdot \text{in}_1))$
 $\quad \quad \quad , \text{in}_2(\underline{K}(\text{fix}_{ct''})(\text{out}_2 \cdot G \cdot \text{in}_2))$

- if $ct = rt \rightarrow rt'$ then

$$\lambda occ. [goto \ell_1] \wedge [def \ell_2] \wedge (G(\lambda occ'. [call(\ell_2)]) occ^{\wedge[1]} [rtn]^{\wedge[def(\ell_1)]^{\wedge[call(\ell_2)]}}$$

where ℓ_1 and ℓ_2 are labels depending uniquely on occ .

The case of pure types is immediate and the case $rt \rightarrow rt'$ is just the usual implementation technique. The case $ct \times ct'$ should be understandable to any one familiar with Bekic's theorem and the case $ct' + ct$ is analogous. We have no ideas of how to handle $rec\ X.ct'$ and $ct' \rightarrow ct$ and we therefore define TMLsc to be TMLs except that all fix_{ct} occurring in expressions must have ct to be composite. The code generated is not always as nice as would be desirable, but in this paper we do not have the space to perform the improvements (Nielson and Nielson, 1985).

THE CORRECTNESS PREDICATE

Let e be an expression of type $rt \rightarrow rt'$ and with no free variables. Correctness of $g = \underline{K}[[e]]$ with respect to $f = \underline{S}[[e]]$ amounts to commutativity of

(abbreviated $cor2(f, g)$). Here $DATA$ and $KONF$ are flat domains of data elements (i.e. values on st) and (st, cs) pairs. The function $init = strict(\lambda r. ([r], []))$ constructs the initial configuration.

Representation of abstract values by concrete values is handled by the function $R[[rt]]$ that is defined structurally on rt . When rt is $\underline{A}_1$ it is an a priori given function. When rt is $rt' \times rt''$ it is of the form $\pi \cdot \lambda(u', u''). (R[[rt']](u'), R[[rt'']](u''))$ where $\pi: DATA^2 \rightarrow DATA$ is some way of representing tuples. We cannot be more precise about π without going into the details of $DATA$ (but one could for example use the boxed data structures of Cardelli. The case $rt' + rt''$ is similar and for the case $\underline{X}$ we need to consider

$$R(rep: D \rightarrow DATA) [[rt]] : \underline{S}[[rt]](D) \rightarrow DATA$$

instead of $R[[rt]]$ and then $R(rep) [[\underline{X}]] = rep$. Finally,

$$R[[\underline{rec}\ X.rt]] = \underline{u}_n (\lambda rep. R(rep) [[rt]])^n(1) \cdot r_n^u$$

where r_n are the embeddings of the limiting cone of the chain $\{\cdot, \underline{S}[[rt]](\cdot), \dots$

The idea is to view $(\lambda rep. R(rep) [[rt]])^n(1)$ as a cone of this chain and then use the formula for the unique mediating morphism (Smyth and Plotkin, 1982).

Unfortunately, we shall need to assert more properties. One set of properties concerns the way $(g\ occ)$ may depend upon occ . The idea is that it is independent of

occ except for an obvious dependency of the labels upon occurrences. Furthermore, that any label used is also defined. Let $\text{lab}(g)$ be the precise formulation of this kind of properties. It will imply that e.g. $\text{RUN}(g[],1) = \text{RUN}(g \text{ occ},1)$. The second set of properties express information that in some cases can be obtained from $\text{cor2}(f,g)$ but it is handy to have it available directly. So let $\text{wdf}(g)$ mean $\text{lab}(g)$ and

$$\text{RUN}(g[],1) ([],[]) \in \{([],[]), \perp, \text{error}_{\text{st}}\}$$

$$\text{RUN}(g[],1) ([r]^{\text{st}}, \text{cs}) \in \{([r']^{\text{st}}, \text{cs}), \perp, \text{error}_r \mid r' \in \text{DATA}\}.$$

In particular error_{cs} cannot occur, i.e. it will never arise that a rtn is to be executed and the control stack is empty.

Consider next a general expression e of type ct_0 and with free variables x_i of type ct_i . The predicates cor2 , lab , and wdf can be extended to apply to $\text{ct} = \text{ct}_1 \times \dots \times \text{ct}_n \rightarrow \text{ct}_0$ using the idea of a relational functor (Reynolds). For the present purposes this amounts to defining a predicate

$$\text{sim}_{\text{ct}}(Q)[\text{sm}]: \underline{S}[[\text{ct}]](D) \times \underline{K}[[\text{ct}]](E) \rightarrow \{\text{true}, \text{false}\}$$

where Q is a predicate upon $D \times E$ and $\text{sm}_{\text{rt} \rightarrow \text{rt}'}$ is a predicate upon $\underline{S}[[\text{rt} \rightarrow \text{rt}']] \times \underline{K}[[\text{rt} \rightarrow \text{rt}']]$. The definition is

$$\text{sim}_A(Q)[\text{sm}](u,v) \equiv u=v$$

$$\text{sim}_{\text{ct}' \times \text{ct}''}(Q)[\text{sm}](u,v) \equiv \text{sim}_{\text{ct}'}(Q)[\text{sm}](u \downarrow 1, v \downarrow 1) \wedge \dots$$

$$\text{sim}_{\text{ct}' + \text{ct}''}(Q)[\text{sm}](u,v) \equiv \exists u', v':$$

$$(u = \text{in}_1(u') \wedge v = \text{in}_1(v') \wedge \text{sim}_{\text{ct}'}(Q)[\text{sm}](u', v')) \vee \dots$$

$$\text{sim}_{\text{ct}' \rightarrow \text{ct}''}(Q)[\text{sm}](u,v) \equiv \forall u', v':$$

$$\text{sim}_{\text{ct}'}(Q)[\text{sm}](u', v') \Rightarrow \text{sim}_{\text{ct}''}(Q)[\text{sm}](u(u'), v(v'))$$

$$\text{sim}_X(Q)[\text{sm}](u,v) \equiv Q(u,v)$$

$$\text{sim}_{\text{rec } X.\text{ct}}(Q)[\text{sm}](u,v) \equiv \forall n. \text{SIM}_n(r_n^u(u), s_n^u(v))$$

$$\text{where } \text{SIM}_0(u,v) \equiv \text{true}, \text{SIM}_{n+1}(u,v) \equiv \text{sim}_{\text{ct}}(Q, \text{SIM}_n)[\text{sm}](u,v)$$

$$\text{and } r_n, s_n \text{ are the appropriate embeddings}$$

$$\text{sim}_{\text{rt} \rightarrow \text{rt}'}(Q)[\text{sm}](u,v) \equiv \text{sm}_{\text{rt} \rightarrow \text{rt}'}(u,v).$$

Then the correctness of $\underline{K}[[e]]$ with respect to $\underline{S}[[e]]$ may be taken to be

$$\text{sim}_{\text{ct}}() [\lambda(f,g). \text{cor2}(f,g) \wedge \text{wdf}(g)] (\underline{S}[[e]], \underline{K}[[e]]).$$

A better possibility is something like

$$\text{sim}_{\text{ct}}() [\text{wdf}] (\underline{K}[[e]]) \wedge \text{sim}_{\text{ct}}() [\lambda(f,g). \text{cor2}(f,g) \wedge \text{wdf}(g)] (\underline{S}[[e]], \underline{K}[[e]])$$

since it allows a more modular proof.

However, one problem remains. The proof of correctness for $\text{fix}_{\text{rt} \rightarrow \text{rt}'}$ is such

that when considering $\underline{K}(\text{fix})(G)$ it is necessary to relate the effects of (Gg_1) and (Gg_2) given a relation between g_1 and g_2 . A useful tool for this is:

Substitution Property $(Gg)[\] = \phi_1 \wedge (g \text{ occ}_1) \wedge \dots \wedge \phi_n$

where the ϕ_i and occ_i are independent of g . ■

The problem then is how to achieve that this property holds. To avoid an ad hoc proof we proceed using a modification of the idea of a relational functor.

First, let "special occ" be a new (dummy) instruction. Define $\text{subst}[h](g_1, g_2)$ by

1) no "special occ" in $(g_1[\])$

2) $(g_1[\]) = (g_2[\])[h \text{ occ}/\text{special occ}]$

with $[a_1; \dots; [b_1; \dots]/c] = [a_1; \dots; a_{i-1}; b_1; \dots; a_{i+1}; \dots]$ when $a_i = c$

i.e. that $(g_1[\])$ is $(g_2[\])$ with $(h \text{ occ})$ expanded in-line in place of "special occ". (This definition can be extended to apply also to the improved code generation that was mentioned earlier.) We do not have the space to prove:

Lemma. The substitution property follows from

$\forall h: \text{sim}_{\text{ct}}()[\text{subst}[h]](G, G)$ ■

(The idea is to exploit $\text{subst}[h](h, \lambda \text{occ}.\{\text{special occ}\})$.)

To summarize, it seems helpful to expand the relational functor with a universal quantifier. The required correctness predicate P_{ct} upon $F \in \underline{S}[[\text{ct}]]$ and $G \in \underline{K}[[\text{ct}]]$ then is (omitting indices)

$P(F, G) \equiv \text{sim}()[\text{lab}](G) \wedge \text{sim}()[\text{wdf}](G)$

$\wedge \forall h: \text{sim}()[\text{subst}[h]](G, G)$

$\wedge \forall h: \text{sim}()[\lambda(f, g_1, g_2). \text{wdf}(g_1) \wedge \text{subst}[h](g_1, g_2) \wedge \text{cor } 2(f, g_1)](F, G, G)$

where all but the last case is motivated by the desire to allow for a modular proof.

THE CORRECTNESS PROOF

The proof of $P_{\text{ct}}(\underline{S}[[e]], \underline{K}[[e]])$ proceeds by structural induction on e except that correctness of the constants, i.e. $P(\underline{S}(f_1), \underline{K}(f_1))$, will be required rather than proved. Let us consider the case $e'oe''$ and let $f = \underline{S}(\Box)(f', f'') = f' \cdot f''$ and $g = \underline{K}(\Box)(g', g'') = \lambda \text{occ}.(g' \text{occ}^{\wedge[2]})^{\wedge}(g' \text{occ}^{\wedge[1]})$. We may assume $P(f', g')$ and $P(f'', g'')$ and must prove $P(f, g)$. It is straight-forward to show that $\text{lab}(g')$ and $\text{lab}(g'')$ imply $\text{lab}(g)$. Also that $\text{subst}[h](g, g)$ follows from $\text{subst}[h](g', g')$ and $\text{subst}[h](g'', g'')$. Next assume $\text{wdf}(g')$ and $\text{wdf}(g'')$. Define the following transformation on configurations

$\tau_n(\text{pc}, \text{st}, \text{cs}) = (\text{pc}+n, \text{st}, [(\text{cs} \downarrow 1)+n; \dots; (\text{cs} \downarrow \text{length}(\text{cs}))+n])$

and the following simulation relation on programs

$\phi' \succ_n \phi$ iff $\forall (pc, st, cs), \gamma$:

- 1) $(pc, st, cs) \xrightarrow{\phi} \gamma \Rightarrow \tau_n(pc, st, cs) \xrightarrow{\phi} \tau_n(\gamma)$ and
- 2) $(pc, st, cs) \not\xrightarrow{\phi} \wedge \text{length}(\phi') \leq \text{length}(\phi)$

$\Rightarrow \tau_n(pc, st, cs) \not\xrightarrow{\phi}$ and giving the same error (see RUN).

The idea is that ϕ' is a shorter program than ϕ and as long as ϕ' can execute also ϕ can, and in a similar way. The first of the two conditions in the definition corresponds to Milner's (1971) notion of a weak simulation whereas the addition of the second condition gives some of the flavour of Milner's notion of a strong simulation. It is straightforward that $g'[] \succ_0 g[]$ and $g''[] \succ_n g[]$ where n is the length of $g'[]$. From this we get

$$\text{RUN}(g[], 1) = \text{RUN}(g''[], 1) \cdot \text{RUN}(g'[], 1)$$

and $\text{wdf}(g)$ follows. Using the above equation we also fairly easily get $\text{cor2}(f, g)$ from $\text{cor2}(f', g')$ and $\text{cor2}(f'', g'')$. This completes the proof for $e' \text{De}$ and most other cases resemble this proof.

Finally we consider fix . So we assume $P_{\text{ct} \rightarrow \text{ct}}(F, G)$ and are to show $P_{\text{ct}}(\underline{S}(\text{fix})(F), \underline{K}(\text{fix})(G))$ where $\underline{S}(\text{fix})$ is the least fixed point operator. The proof is by cases on the composite type ct . One base case is when ct is pure and then the result is easy. If $\text{ct} = \text{ct}' \times \text{ct}''$ the result follows from the induction hypotheses and Bekic's Theorem. Similar remarks apply to $\text{ct} = \text{ct}' + \text{ct}''$ so we are left with the second base case $\text{ct} = \text{rt} \rightarrow \text{rt}'$. (This base case is more complicated when considering the improved code generation mentioned earlier.) The main idea is to extend the machine with a counter corresponding to n in $\bigcup_{n=0}^{\infty} F^n(1)$; in this way " $=$ " between $\underline{S}$ and $\underline{K}$ can be proved directly rather than by separate proofs of " $\sqsubseteq$ " and " $\sqsupseteq$ ".

A configuration in the extended machine is a triple (pc, st, cs) where cs is a list of pairs (ℓ, pc') of labels and program counters. A " $\text{call}(\ell)$ " then places $(\ell, pc+1)$ rather than just $pc+1$ on top of cs . Analogously the rewrite relation $\rightarrow$ is indexed by a finite set Δ of pairs (ℓ, n) . It mostly behaves like $\rightarrow$. However, suppose Δ contains (ℓ, n) and cs contains more than n copies of ℓ . Then $(pc, st, cs) \xrightarrow{\Delta} (pc, st, cs)$ contrary to the behavior of $\rightarrow$. Let RUN_{Δ} be defined similarly to RUN . Then $\text{RUN}_{\Delta}(\{(\ell, n)\}(\phi, 1)(st, []))$ will terminate only if at most n calls of ℓ will be active at any point in the computation. (This is analogous to the behaviour of $F^{n+1}(1)$.)

It is not too hard to show $\text{lab}(\underline{K}(\text{fix})(G))$ and $\text{subst}[h](\underline{K}(\text{fix})(G), \underline{K}(\text{fix})(G))$ so let us consider wdf and cor2 . Define $\mathbb{1} = \lambda \text{occ}. [\text{loop}]$ where " loop " is some new instruction that loops. (Alternatively one could use the existing constructions to build a program with the same effect.) The proof proceeds by showing the following 5 results:

$$\text{Res 1} \quad \forall n: \text{cor2}(F^n \mathbb{1}, G^n \mathbb{1}) \wedge \text{wdf}(G^n \mathbb{1})$$

- Res 2 $\forall n: \text{RUN}_{\emptyset}(G^{\mathbf{n}} \mathbb{L}[], l)(st, []) = \text{RUN}_{\{(\ell_2, n)\}}(\underline{K}(\text{fix})(G) [], l)(st, [])$
 where ℓ_2 is as in the definition of $\underline{K}(\text{fix})(G)$
- Res 3 $\text{RUN}_{\emptyset}(\phi, l) = \bigcup_n \text{RUN}_{\{(\ell_2, n)\}}(\phi, l)$
- Res 4 $\text{strip} \cdot \text{RUN}_{\emptyset}(\phi, l) = \text{RUN}(\phi, l) \cdot \text{strip}$
 where $\text{strip}(pc, st, [(\ell_1, pc_1); \dots]) = (pc, st, [pc_1; \dots])$
- Res 5 $\text{cor2}(\underline{S}(\text{fix})(F), \underline{K}(\text{fix})(G)) \wedge \text{wdf}(\underline{K}(\text{fix})(G))$

Apart from Res 2 the proofs of these results are relatively straightforward: Res 1 is a simple induction, Res 3 is a simple induction on the number of steps of the machine, Res 4 is intuitively obvious (or use the simulation idea), and Res 5 follows from continuity of $\mathcal{R}[[rt]]$ and init .

It is Res 2 that is the key in relating the recursive program to its unfoldings. The result is proved by induction on n and the case $n=0$ is trivial, so consider $n=m+1>0$. Using the substitution property we have:

$$G^{m+1} \mathbb{L}[] = \phi_0 \wedge (G^m \mathbb{L} \text{occ}_1) \wedge \dots \wedge \phi_k$$

$$\underline{K}(\text{fix})(G)[] = [\text{goto } \ell_1] \wedge [\text{def } \ell_2] \wedge \phi_0 \wedge [\text{call } \ell_2] \wedge \dots \wedge \phi_k \wedge [\text{rtn}] \wedge [\text{def } \ell_1] \wedge [\text{call } \ell_2]$$

One can then define a correspondence between instructions of the two programs. The second program starts with a few sequencing steps. Then they execute corresponding instructions of ϕ_0 (or in general some ϕ_i) maintaining a certain relation between the configurations. When one program goes into $G^m \mathbb{L} \text{occ}_1$ and the other makes a "call ℓ_2 " the induction hypothesis is used so that one can continue with the following ϕ_{i+1} . This ends the informal proof of Res 2; the formal proof is precise about the relation between corresponding configurations and is therefore quite a bit longer.

CONCLUSION

This paper focused on the correctness of code generation from a two-level meta language. The idea of using two levels is believed to be profound, and it is hoped that TMLsc is sufficiently general to be useful. Perhaps its main limitation is the one inherited from TMLs, namely that $rt := rt' \rightarrow rt$ is absent; if this is to be handled it is likely that the correctness predicate must be changed so as to avoid $\mathcal{R}[[rt' \rightarrow rt]]$. At any rate TMLsc is sufficiently general that the correctness proof is complex. The claim then is that this paper presents worth-while techniques for reducing the complexity to an acceptable level. Further it is claimed that the techniques are not particular to the actual code generation $\underline{K}$ considered, i.e. that the proofs for other code generation interpretations will have the same structure as for $\underline{K}$. (In the case of abstract interpretation theorem 3.3:14 of (Nielson, 1984) shows such a result.)

ACKNOWLEDGEMENT

This research is part of the PSI-project which is partly supported by the Danish Natural Science Research Council.

REFERENCES

Backus, J., Can Programming Be Liberated from the von Neuman Style, CACM Vol. 21 No. 8 (1978).

Cardelli, L., The Functional Abstract Machine, Bell Labs Technical Report TR-107 (1983).

Cousot, P., and Cousot, R., Systematic Design of Program Analysis Framework, 6th POPL (1979).

Dybjer, P., Using Domain Algebras to Prove the Correctness of a Compiler, STACS 1985, Springer Lecture Notes in Computer Science.

Gordon, M., The Denotational Description of Programming Languages, Springer Verlag (1979).

Milner, R., and Strachey, C., A Theory of Programming Language Semantics, Chapman and Hall (1976).

Milner, R., An Algebraic Definition of Simulation between Programs, in: Proceedings of 2nd International Conference on Artificial Intelligence, British Comp. Soc. (1971).

Morris, F.L., Advice on Structuring Compilers and Proving Them Correct, POPL 1973, pp. 144-152.

Mosses, P.D., A Constructive Approach to Compiler Correctness, ICALP 1980, Springer Lecture Notes in Computer Science.

Nielson, F., Abstract Interpretation using Domain Theory, Ph.D. thesis, University of Edinburgh (1984).

Nielson, H.R., and Nielson, F., Pragmatic Aspects of Two-level Metalanguages, these proceedings (1986).

Nielson, F., and Nielson, H.R., Code Generation from Two-level Denotational Meta-Languages, in the proceedings from "Programs as Data Objects", (Copenhagen 1985), Springer Lecture Notes in Computer Science.

Reynolds, J., On the Relation between Direct and Continuation Semantics, ICALP 1974, Springer Lecture Notes in Computer Science, vol. 14.

Smyth, M., and Plotkin, G.D., The Category-Theoretic Solution of Recursive Domain Equations, SIAM J. Comput., Vol. 11 No. 4 (1982).

Tennent, R.D., Principles of Programming Languages, Prentice-Hall (1981).

Thatcher, J.W., and Wagner, E.G., and Wright, J.B., More on Advice on Structuring Compilers and Proving Them Correct, 6th ICALP (1979).