PRAGMATIC ASPECTS OF TWO-LEVEL DENOTATIONAL META-LANGUAGES

Hanne R. Nielson
Flemming Nielson

ABSTRACT

This work is part of a research project on automatic generation of optimazing compilers from denotational language definitions. The novel aspect of our approach is that we are based on a two-level meta-language allowing us to distinguish between compile-time and run-time, and thereby to formalize e.g. the distinction between "static expression procedures" and "expression procedures" of Tennent (1981). In this paper we discuss some of the problems encountered when writing denotational definitions using a two-level meta-language. We consider the meta-language TML_s introduced in Nielson (1986a) as well as its restricted version TML_{sc} developed in Nielson and Nielson (1986) for automatic code generation. Based on an example we argue that rewriting a language definition using TML_s in TML_{sc} really means introducing some notion of activation record. This observation may pave the way for a formalization of the transformations on semantic definitions considered by Milne and Strachey (1976) as being imposed by different meta-languages.

1. INTRODUCTION

It is well-known that the distinction between compile-time and run-time is important for the efficient implementation of programming languages. In standard denotational definitions (see Gordon (1979), Stoy (1977)) there is no such distinction and we believe this to be a serious reason for the lack of success of the formalism in automatic compiler writing systems. None the less, a lot of effort has been made in this area, see e.g. Appel (1985), Jones and Christiansen (1982), Paulson (1984), Sethi (1983) and Wand (1982).

On the other hand, attribute grammars have been successful in compiler construction (see e.g. Kastens (1984) and Räiha (1984)) mainly because they can be implemented reasonably efficiently. Attribute grammars can be used for specifying static semantics and code generation, that is, the compile-time actions of a compiler. The semantic grammars of Paulson (1984) combine the attribute grammars and the denotational semantics by using the latter for specification of dynamic semantics and thereby the run-time actions of the compiler. So Paulson's formalism allows a dis-

---

Authors' address: Institute of Electronic Systems, Aalborg University Centre, Strandvejen 19, 4, DK-9000 Aalborg C, Denmark

tinction between binding times although it is rather undeveloped. Our two-level meta-languages formalise this distinction.

In traditional compilers data flow analysis and program transformations are used to improve the generated code. Such approaches have been applied in MUG2, an attribute grammar based system for automatic compiler generation (Ganzinger et al, 1982). Recent work (Nielson 1984, 1986a) on abstract interpretation, a framework for specifying data flow analysis and proving it correct, paves the way for a systematic treatment of data flow analyses and program transformations in the realm of (two-level) denotational semantics. In Nielson and Nielson (1986) the framework has been extended to include code generation as well.

The aim of this paper is to discuss some of the problems encountered when writing denotational definitions using a two-level meta-language.

2. THE META-LANGUAGE TML_s

Traditional denotational definitions (Stoy (1977), Gordon (1979)) use a typed λ-calculus as meta-language. In a two-level meta-language we want to distinguish between compile-time entities and run-time entities, and this is accomplished by introducing two sorts of types, compile-time types (ct) and run-time types (rt). The type structure could for instance be given by:

$$ct ::= A \mid ct_1 \times \ldots \times ct_k \mid ct_1 + \ldots + ct_k \mid \text{rec } X.ct \mid X \mid ct_1 \rightarrow ct_2' \mid rt$$

$$rt ::= \underline{A} \mid rt_1 \underline{\times} \ldots \underline{\times} rt_k \mid rt_1 \underline{+} \ldots \underline{+} rt_k \mid \underline{\text{rec } X}.rt \mid \underline{X} \mid rt_1 \underline{\rightarrow} rt_2$$

The syntaxes of the two levels are rather similar so underlining is used to disambiguate. We use $A(\underline{A})$ to denote base types and assume that the truth values $T(\underline{T})$ are included. Compile-time types are combined using the k-ary cartesian product $\times$, the k-ary discriminated union + and they can be recursively defined via rec X.ct and the possibility of using X in ct. Finally, function spaces can be constructed as in $ct_1 \rightarrow ct_2$. The interpretation of the run-time type constructors may be varied to obtain different effects. As we shall return to later, the standard interpretation of $\underline{\times}$ will be smash product, $\underline{+}$ will be coalesced sum and $\underline{\rightarrow}$ will be strict function space. The interaction between the two type levels is restricted by the absense of rt::= ct. This reflects the intuition that compile-time entities cannot be discussed at run-time. The presence of ct::= rt allows us to handle run-time entities (such as $rt_1 \underline{\rightarrow} rt_2$ which may be viewed as code) at compile-time. Extending the type structure above with an appropriate expression language gives a very powerful meta-language.

Abstract interpretation, a framework for describing and proving correctness of certain data flow analyses, has been developed for a subset TML_s of this meta-language, Nielson (1986a, 1984). Technical problems make it hard to deal with the run-

time types $rt_1 \underline{\rightarrow} rt_2$ in full generality so the type structure of TML_s is somewhat restricted:

$$ct ::= A \mid ct_1 \times \ldots \times ct_k \mid ct_1 + \ldots + ct_k \mid \text{rec } X.ct \mid X \mid ct_1 \rightarrow ct_2 \mid rt_1 \underline{\rightarrow} rt_2$$

$$rt ::= \underline{A} \mid rt_1 \underline{\times} \ldots \underline{\times} rt_k \mid rt_1 \underline{+} \ldots \underline{+} rt_k \mid \underline{\text{rec } X}.rt \mid \underline{X}$$

Intuitively, this means that functions cannot be treated as other data objects; in Strachey's terminology they are not "first class", and we express this by calling them "second class". The framework for codegeneration developed in Nielson and Nielson (1986) (and reviewed in Section 4) is based on TML_s as well, but may be easier to extend to the general meta-language although the correctness proof does not easily generalize (Nielson 1986b).

In view of these technical problems we shall mainly be interested in TML_s which, despite of its limitations, still makes it possible to define the semantics of a large class of imperative languages.

Turning to the expression language of TML_s it is defined as follows:

$$\begin{aligned} e ::= {} & f \mid (e_1,\ldots,e_k) \mid e \downarrow j \mid in_j e \mid is_j e \mid out_j e \\ & \mid \lambda x\text{: } ct.e \mid e_1(e_2) \mid x \mid \text{mkrec } e \mid \text{unrec } e \\ & \mid e \rightarrow e_1, e_2 \mid fix_{ct}\ e \\ & \mid \underline{\text{tuple}}\ (e_1,\ldots,e_k) \mid \underline{\text{take}}_j \mid \underline{\text{in}}_j \mid \underline{\text{case}}\ (e_1,\ldots,e_k) \\ & \mid \underline{\text{mkrec}} \mid \underline{\text{unrec}} \mid \underline{\text{cond}}\ (e,e_1,e_2) \mid e_1 \square e_2 \end{aligned}$$

Here f denotes a constant of type ct but not all compile-time types ct are allowed: ct must be contravariantly pure and closed (Nielson, 1984). The idea is that there must not be run-time function spaces in the domain of a compile-time function space in ct and in addition ct must not contain free (type) variables. Furthermore, a number of natural typing restrictions are imposed to ensure that the expressions are well-typed (Nielson, 1984).

The first part of the notation for expressions is quite standard and is explained by e.g. Stoy (1977). The second part is more "algebraic" in the style of e.g. Backus (1978), Mosses (1982), Rault and Sethi (1983) and relates to the run-time level of the type system.

The semantics of the meta-language is given by an interpretation $\underline{I}$ consisting of two parts, a type part and an expression part. The type part will define a cpo $\underline{I}[\![ct]\!]$ for each closed type ct (Nielson, 1984). In the case of the standard interpretation $\underline{S}$, the cpo's are obtained in the traditional way by interpreting $\times$ as cartesian product, $\underline{\times}$ as smash product, + as separated sum, $\underline{+}$ as coaleased sum, $\rightarrow$ as function space and $\underline{\rightarrow}$ as strict function space. Recursive domain equations (rec X.ct and $\underline{\text{rec } X}.rt$) are solved up to isomorphism using the categorical approach of Smyth and Plotkin (1982).

The expression part of the interpretation defines a function $\underline{I}[\![e]\!] : \underline{I}[\![ct_1]\!] \times \ldots \times \underline{I}[\![ct_n]\!] \to \underline{I}[\![ct]\!]$ for each well-typed expression e (with free variables of closed types $ct_1,\ldots,ct_n$, resp., and result of type ct). We shall omit the detailed typing conventions (see Nielson (1984)). The λ-calculus part of the expression language is interpreted as usual; in particular mkrec and unrec are the isomorphisms Θ and Θ^{-1} obtained from solving recursive domain equations. In the standard interpretation $\underline{S}$ the combinator style expressions are interpreted as follows:

$\underline{S}(\underline{tuple}): \underline{S}[\![rt \to rt_1]\!] \times \ldots \times \underline{S}[\![rt \to rt_k]\!] \to \underline{S}[\![rt \to rt_1 \times \ldots \times rt_k]\!]$

$\underline{S}(\underline{tuple}) = \lambda(E_1,\ldots,E_k).\ \lambda v.\ (E_1 v,\ldots,E_k v)$

$\underline{S}(\underline{take}_j): \underline{S}[\![rt_1 \times \ldots \times rt_k \to rt_j]\!]$

$\underline{S}(\underline{take}_j) = \lambda v.\ v \downarrow j$

$\underline{S}(\underline{in}_j): \underline{S}[\![rt_j \to rt_1 + \ldots + rt_k]\!]$

$\underline{S}(\underline{in}_j) = \lambda v.\ in_j v$

$\underline{S}(\underline{case}): \underline{S}[\![rt_1 \to rt]\!] \times \ldots \times \underline{S}[\![rt_k \to rt]\!] \to \underline{S}[\![rt_1 + \ldots + rt_k \to rt]\!]$

$\underline{S}(\underline{case}) = \lambda(E_1,\ldots,E_k).\ \lambda v.\ is_1 v \to E_1(out_1 v),\ldots,$
$is_k v \to E_k(out_k v), \perp$

$\underline{S}(\underline{mkrec}): \underline{S}[\![rt[\underline{recX}.rt/\underline{X}] \to \underline{recX}.rt]\!]$

$\underline{S}(\underline{mkrec}) = \Theta$

$\underline{S}(\underline{unrec}): \underline{S}[\![\underline{recX}.rt \to rt[\underline{recX}.rt/\underline{X}]]\!]$

$\underline{S}(\underline{unrec}) = \Theta^{-1}$

$\underline{S}(\underline{cond}): \underline{S}[\![rt \to \underline{T}]\!] \times \underline{S}[\![rt \to rt']\!] \times \underline{S}[\![rt \to rt']\!] \to \underline{S}[\![rt \to rt']\!]$

$\underline{S}(\underline{cond}) = \lambda(E,E_1,E_2).\ \lambda v.\ Ev \to E_1 v, E_2 v$

$\underline{S}(\square): \underline{S}[\![rt_2 \to rt_3]\!] \times \underline{S}[\![rt_1 \to rt_2]\!] \to \underline{S}[\![rt_1 \to rt_3]\!]$

$\underline{S}(\square) = \lambda(E_1,E_2).\ \lambda v.\ E_1(E_2 v)$

The motivation for letting the expressions relating to the run-time types focus on functions rather than elements stems from applications in abstract interpretation as well as code generation where "run-time state transformations" and "code" are of main interest. To some extent one could device an automatic translation of a more standard notation like that of Stoy (1977) into this "algebraic" style.

3. SEMANTICS OF SMALL USING TML_s

Writing denotational definitions using a two-level meta-language as TML_s rather than the traditional meta-language of e.g. Gordon (1979) and Stoy (1977) may pose some problems especially because we have to distinguish between compile-time and

run-time. In this section we shall report on a rewriting of the definition of the toy language SMALL, introduced by Gordon (1979).

The main syntactic categories of SMALL are expressions (E), commands (C), and declarations (D). Because of the limited space we shall mainly be interested in the declarations:

D::= const I=E | var I=E | proc $I(I_1)$; C | fun $I(I_1)$; E | D_1; D_2

Since the meta-language TML_s prohibits functions as first class data objects we shall restrict the language of Gordon (1979) and disallow functions and procedures as values of expressions.

When rewriting a traditional denotational definition in a two-level meta-language it may be helpful with some guidelines for how to distinguish between compile-time and run-time domains. Intuitively,

- environments (Env = Ide → Dv) and denotable values (Dv) are compile-time domains,
- stores or states (S = Loc → Sv) and storable values (Sv) are run-time domains,
- expressible values (Ev) and R-values (Rv) are run-time values.

The semantics of Gordon (1979) is rather unprecise with respect to the handling of static expressions (const I=E) and expressions (var I=E). In both cases E is evaluated to give an Rv-element, which in the first case is bound in Env and in the second case in S. Thus Rv is used as a compile-time domain in Dv=...+Rv+... and as a run-time domain in Sv=...+Rv+... The solution, of course, is to have two versions of Rv, one for the compile-time level and one for the run-time level.

The denotations of procedures (Proc) and functions (Fun) provide a link between the compile-time and the run-time levels of the semantics in that they, essentially, express a state transformation.

Using the domains of Table 1 below the semantic functions have the following functionalities:

$\mathcal{E},\mathcal{R}$: Exp → Env → Loc → State → Ev × State

$\mathcal{S}$: Exp → Env → Rv + ϕ

$\mathcal{C}$: Com → Env → Loc → State → State

$\mathcal{D}$: Dec → Env → Loc → Env × Loc × (State → State)

TABLE 1

Basic run-time domains: $\underline{N}$, $\underline{T}$, $\underline{\phi}$

Compound run-time domains:

$\underline{Loc} = \underline{N}$	$\underline{File} = \underline{recF.\ \phi + Rv \times F}$
$\underline{Rv} = \underline{T + N}$	$\underline{Sv} = \underline{Rv}$
$\underline{Ans} = \underline{recA.\ \phi + \phi + Rv \times A}$	$\underline{Mem} = \underline{recM.\ \phi + (Sv+\phi) \times M}$
$\underline{Ev} = \underline{Loc + Rv + \phi}$	$\underline{State} = \underline{Mem \times File \times Ans}$

Basic compile-time domains: Ide, N, T, ϕ

Compound compile-time domains:

$Loc = N$	$Proc = Loc \to \underline{Ev \times State \to State}$
$Rv = T + N$	$Dv = Loc + Rv + Proc + Fun$
$Fun = Loc \to \underline{Ev \times State \to Ev \times State}$	$Env = Ide \to Dv + \phi$

For constant and variable declaration we get the clauses:

$$\mathcal{D}[\![\text{const I=E}]\!]\ r\ell = (\text{upd-env}(\text{val}(\mathcal{S}[\![\text{E}]\!]\ r), I, r),\ \ell, \text{Id}_{\underline{State}})$$

$$\mathcal{D}[\![\text{var I=E}]\!]\ r\ell = (\text{upd-env}(\text{in}_1(\text{in}_1\ell), I, r),\ \text{next-loc}\ \ell,$$
$$\text{upd-state} \square \underline{\text{tuple}}\ (\text{conv-loc}\ \ell \square \underline{\text{take}_2}, \text{Id}_{\underline{Ev \times State}}) \square \mathcal{R}[\![\text{E}]\!]\ r\ell)$$

These two clauses express very precisely the difference between the static expression and the expression in the declarations. The function conv-loc: Loc → $\underline{State \to Loc}$ transforms a compile-time location into a run-time location. Note that the type structure of TML_s prevents us from having a direct transformation from Loc to $\underline{Loc}$. The function val: Rv+ϕ → Dv+ϕ is injection whereas upd-env: (Dv+ϕ)× Ide × Env → Env updates an environment, upd-state: $\underline{Loc} \times (\underline{Ev} \times \underline{State}) \to \underline{State}$ updates a state and next-loc: Loc → Loc gives the next free location. We omit the detailed definitions.

Because functions may be recursive we get the rather complicated clause (explained below):

$\mathcal{D}[\![\text{fun I}(I_1); \text{E}]\!]\ r\ell =$

(1) (upd-env(in$_1$(in$_4$(fix$_{Fun}$(λf:Fun.$\lambda\ell'$:Loc.

(2) $\underline{\text{tuple}}(\underline{\text{take}_1}$,pop-state ℓ' □ $\underline{\text{take}_2})$

(3) □ $\mathcal{E}[\![\text{E}]\!]$ upd-env(in$_1$(in$_1\ell'$),I_1,upd-env(in$_1$(in$_4$f),I,r))(next-loc ℓ')

(4) □ upd-state □ $\underline{\text{tuple}}$ (conv-loc ℓ' □ $\underline{\text{take}_2}$,Id$_{\underline{Ev \times State}}$))))),

(5) I,r),

(6) ℓ,Id$_{\underline{State}}$)

Line (1)-(5) updates the environment with the function declaration whereas (6) records that the next free location is unchanged and that no state transformation has occurred. The meaning of the function is denoted by f in line (2)-(4) and is (recursively) defined by these lines: The next free location when the function is called is denoted ℓ' and in line (3) the formal parameter I_1 is bound to that location in the environment (and the function name I is bound to f). In line (4) the actual parameter is bound to the location ℓ' in the state (given by conv-loc ℓ'). Line (3) expresses the effect of executing the function body in the state so obtained. Finally, in line (2) the binding of values to ℓ' in the state is undone (using pop-state ℓ': State → State).

The semantic functions defined in this way correspond closely to those obtained by using the technique of continuation removal (Milne and Strachey (1976), Stoy (1977)) on the semantic functions of Gordon (1979). Some problems arise when rewriting a standard continuation style semantics in TML_s. Consider the traditional clause:

$$\mathcal{E}[\![E_1+E_2]\!]\ rk = \mathcal{R}[\![E_1]\!]\ r\ (\lambda e_1{:}Ev.\mathcal{R}[\![E_2]\!]\ r\ (\lambda e_2{:}Ev.k(e_1+e_2)))$$

As a result of the rewriting in TML_s the free occurrences of e_1 and e_2 must be removed. Using the idea of Wand (1982) we get:

$$\mathcal{E}[\![E_1+E_2]\!]\ rk = \mathcal{R}[\![E_1]\!]\ r\ (B(\mathcal{R}[\![E_2]\!]\ r, k \square add))$$

where B: (Ec → Cc) × (Ev × Ev × Store → Ans) → Ec and add: Ev × Ev × Store → Ev × Store. Misusing the notation, B can be defined as:

$$B(\alpha,\beta) = \lambda(e_1,s_1)\text{: Ev} \times \text{Store}.\ \alpha(\lambda(e_2,s_2)\text{: Ev} \times \text{Store}.\ \beta(e_1,e_2,s_2))(s_1).$$

However, we have not been able to find a TML_s-expression for B. It is interesting to note that for a continuation style store semantics (Milne and Strachey, 1976) this approach will work (by choosing $B(\alpha,\beta)$ to be $\alpha(\beta)$).

4. CODE GENERATION AND TML_{sc}

As we have seen the two-level meta-language allow us to distinguish between compile-time entities and run-time entities. At the compile-time level of TML_s we cannot talk directly about run-time values - we have to talk about transformations on run-time values (of type $rt_1 \rightarrow rt_2$). Intuitively, such transformations can also be obtained by executing a piece of code on an appropriate abstract machine. Therefore, code generation for TML_s amounts to specifying a new interpretation for the basic expressions of type $rt_1 \rightarrow rt_2$: rather than specifying a function between two domains as in the standard interpretation it is now going to specify a piece of code for an abstract machine.

In Nielson and Nielson (1986) we show how to specify such a coding interpretation for an appropriate abstract machine. Unfortunately, some problems arise for the fixed point operator, fix_{ct}, forcing us to restrict the meta-language. Define a compile-time type ct to be composite if (and only if):

- ct has the form $rt_1 \to rt_2$, or
- ct is pure (i.e. contains no run-time types), or
- ct is a cartesian product or a discriminated union of composite types.

The type Fun(=Loc → Ev × State → Ev × State) considered in Section 3 is not composite whereas Ev × State → Ev × State is.

These restrictions give rise to the definition of the meta-language TML_{sc}: The types of TML_{sc} are as those of TML_s and so are the expressions except that wherever fix_{ct} e occurs ct must be composite.

5. SEMANTICS OF SMALL USING TML_{sc}

The semantic definition of SMALL discussed in Section 3 does not fulfill the restrictions imposed by TML_{sc}. The clause for function declaration uses the fixed point operator fix_{Fun} and, as mentioned above the type Fun is not composite. Intuitively, the appearence of Loc in Fun reflects that every time a function is called new storage cells are needed for its parameter and local variables. The association of variables with addresses happens at the compile-time level of the semantic specification. The violation of the requirements of TML_{sc} intuitively means that there are aspects of the semantics that neither belong purely to the compile-time level nor the run-time level and that the association of variables with addresses is such an aspect.

To get an idea of how the problem can be mended let us briefly review the usual run-time organization for block-structured languages, see Aho and Ullman (1977). Each time a function is called a new activation record is pushed on top of the run-time stack. The activation record contains among other things the values of the local variables and the static link pointing to the activation record of the statically surrounding function. Using this chain it is possible to reference variables knowing the top of the stack, the (static) number of the function where the variable is declared, and its offset. So although the exact address cannot be computed at compile-time it is possible to determine the access path.

Similar ideas can be used to rewrite the semantics of Section 3 in TML_{sc}. At compile-time a variable is associated with a pair (b,o) of block number and offset so we redefine the domain Loc to be Loc = N × N. The run-time domain Loc is left unchanged and, as before, it can be interpretated as the absolute addresses on the stack. The function conv-loc: Loc → State → Loc converts a pair of block number and

offset into an absolute address in the given state. In other words, conv-loc implements the access path.

The analogue of the static links in the stack is modelled by extending the domain Sv of storable values to include locations (interpreted as pointers into the stack itself): Sv = Rv + Loc. With these modifications the domains Proc and Fun can be simplified: Proc = Ev × State → State and Fun = Ev × State → Ev × State. The remaining domains are as in Section 3.

The functionalities of the semantic functions are as in Section 3 and so are most of the semantic clauses. The denotation of a function must ensure that the appropriate static link is pushed on the stack before the function body is evaluated and furthermore that it is popped when the function is left. The two auxiliary functions push-ar: N → State → State and pop-ar: State → State ensure that this happens in the clause for function declaration:

$$
\begin{aligned}
&\mathcal{D}[\![\ \text{fun } I(I_1);E]\!]\ r\ell = \\
&\quad (\text{upd-env}(\text{in}_1(\text{in}_4(\text{fix}_{\text{Fun}}(\lambda f\text{: Fun.pop-ar} \\
&\qquad \square\ \mathcal{E}[\![\ E]\!]\ \text{upd-env}(\text{in}_1(\text{next-bn}(\ell)), I_1, \\
&\qquad\qquad \text{upd-env}(\text{in}_1(\text{in}_4 f), I, r))(\text{next-loc}(\text{next-bn}\ \ell)) \\
&\qquad \square\ \text{upd-state} \square \underline{\text{tuple}}(\text{conv-loc}(\text{next-bn}\ \ell) \square \underline{\text{take}_2},\ \text{Id}_{\underline{\text{Ev} \times \text{State}}}) \\
&\qquad \square\ \underline{\text{tuple}}(\underline{\text{take}_1}, \text{push-ar}((\text{next-bn}\ \ell)\ \downarrow 1) \square \underline{\text{take}_2}))))), \\
&\quad I, r), \\
&\ell, \text{Id}_{\underline{\text{State}}}).
\end{aligned}
$$

Here next-bn(b,o) = (b+1,0) whereas next-loc(b,o) = (b,o+1).

6. CONCLUSION

The starting point for this work has been a two-level meta-language for writing denotational definitions. This allows us to distinguish betweeen the notions compile-time and run-time in language specifications - a distinction being very important for the efficient implementation of programming languages, but also for more abstract understanding of current programming languages (Tennent, 1981). The ultimate goal will, of course, be to use a traditional denotational meta-language and then detect the borderline between the two binding times automatically. In Section 3 we gave some heuristic rules for how to determine this borderline.

Milne and Strachey (1976) note that the semantics of a language should be given by the socalled standard semantics where the meanings of programs are formed from "abstract" objects independent of the representation. They then rewrite this semantics to bring it closer to an actual implementation. The motivation behind their store

semantics is that it should be possible to list all the locations that can be accessed by a program at any time during the execution. In the stack semantics this is further substantiated by the adoption of a strict stack allocation of storage - very much as the one we employed in the previous section. The stack semantics is then the starting point for their development of code generation.

It is interesting to note that whereas Milne and Strachey (1976) choose to develop a semantics with this strict stack discipline we are forced to do so by the meta-language TML_{SC}. Intuitively, each restriction of the meta-language arises from a limited ability to automatically process definitions employing the full meta-language, e.g. to obtain efficient compilers from arbitrary semantic definitions. Having isolated these limited abilities into formally defined restrictions it becomes easier to isolate the subtask of how to pass from one meta-language to another. This may again lead to a more systematic treatment of the kind of development that Milne and Strachey perform.

ACKNOWLEDGEMENT

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

REFERENCES

Aho, A.V. and Ullman, J.D., Principles of Compiler Design, Addison-Wesley (1977).

Appel, A.W., Semantics-directed code generation, 12. POPL, 315-324 (1985).

Backus, J.W., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, CACM 21:8, 613-641 (1978).

Ganzinger, H., Giegerich, R., Möncke, U. and Wilhelm, R., A truly generative semantics-directed compiler generator, SIGPLAN 82 Symposium on Compiler Construction, 172-184 (1982).

Jones, N.D. and Christiansen, H., Control flow treatment in a simple semantics-directed compiler generator, In: Formal description of programming concepts II, (ed. D. Bjørner) North-Holland (1982).

Gordon, M.J.C., The denotational description of programming languages, an introduction, Springer Verlag (1979).

Kastens, U., The GAG-system - a tool for compiler construction, In: Methods and tools for compiler construction, (ed: B. Lorho) Cambridge University Press, 165-182 (1984).

Milne, R. and Strachey, C., A theory of programming language semantics, Halsted Press (1976).

Mosses, P.D., Abstract semantic algebras! In: Formal description of programming concepts II, (ed. D. Bjørner) North-Holland (1982).

Nielson, F., Anstract interpretation using domain theory, Ph.D. thesis, Edinburgh University (1984).

Nielson, F., Abstract interpretation of denotational definitions, to appear in STACS Proceedings, LNCS, Springer Verlag (1986a).

Nielson, F., Correctness of code generation from a two-level meta-language, in these proceedings (1986b).

Nielson, F. and Nielson, H.R., Code generation from two-level denotational meta-languages, to appear in proceedings from workshop on Programs as Data Objects (ed. H. Ganzinger, N.D. Jones), LNCS, Springer Verlag (1986).

Paulson, L., Compiler generation from denotational semantics, In: Methods and tools for compiler construction, (ed. B. Lorho) Cambridge University Press, 219-250 (1984).

Rault, J.-C. and Sethi, R., Properties of a notation for combining functions, JACM 30:3, 595-611 (1983).

Räihä, K.-J., Attribute grammar design using the compiler writing system HLP, In: Methods and tools for compiler construction, (ed. B. Lorho) Cambridge University Press, 183-206 (1984).

Sethi, R., Control flow aspects of semantics directed compiling, ACM TOPLAS 5:4, 554-595 (1983).

Smyth, M.B. and Plotkin, G.D., The category-theoretic solution of recursive domain equations, SIAM J. Comput. 11:4, 761-783 (1982).

Stoy J., Denotational semantics, The MIT Press (1977).

Tennent, R.D., Principles of programming languages, Prentice Hall (1981).

Wand, M., Deriving target code as a representation of continuation semantics, ACM TOPLAS, 4:3, 496-517 (1982).