ISSN 0105-8517

Theoretical Aspects of Semantics-Based Language Implementation

Teoretiske aspekter af Semantik-baseret Sprog-implementation

Flemming Nielson

DAIMI PB – 329
August 1990

ISSN 0105-8517

PB – 329 **F. Nielson: Aspects of Semantics-Based Language Implementation**

Denne afhandling er i forbindelse med de nedenfor anførte, tidligere offentliggjorte afhandlinger af Det Naturvidenskabelige Fakultetsråd ved Aarhus Universitet antaget til offentligt forsvar for den naturvidenskabelige doktorgrad.

Forsvaret finder sted mandag den 19. november, kl. 10.15 i Auditorium G.1, Matematisk Institut, Aarhus Universitet.

'Regler for forsvarshandlingen' kan rekvireres ved Administrationen, Aarhus Universitet.

Aarhus, den 21. august 1990,

Karl Pedersen
Dekan

1. F.Nielson: A Denotational Framework for Data Flow Analysis, *Acta Informatica* **18** 265–287 (23 sider), 1982.
2. F.Nielson: Program Transformations in a Denotational Setting, *ACM Transactions on Programming Languages and Systems* **7** *3* 359–379 (21 sider), 1985.
3. A.Mycroft & F.Nielson: Strong Abstract Interpretation using Power Domains, *Proceedings ICALP 1983*, Springer Lecture Notes in Computer Science **154** 536–547 (12 sider), 1983.
4. F.Nielson: Towards Viewing Nondeterminism as Abstract Interpretation, *Foundations of Software Technology & Theoretical Computer Science* **3** (20 sider), 1983.
5. F.Nielson: Abstract Interpretation of Denotational Definitions, *Proceedings STACS 1986*, Springer Lecture Notes in Computer Science **210** 1–20 (20 sider), 1986.
6. F.Nielson: Tensor Products Generalize the Relational Data Flow Analysis Method, *Proceedings 4'th Hungarian Computer Science Conference* 211–225 (15 sider), 1985.
7. F.Nielson: Expected Forms of Data Flow Analysis, *Programs as Data Objects*, Springer Lecture Notes in Computer Science **217** 172–191 (20 sider), 1986.
8. H.R.Nielson & F.Nielson: Semantics Directed Compiling for Functional Languages, *ACM Conference on LISP and Functional Programming* 249–257 (9 sider), 1986.
9. F.Nielson: Strictness Analysis and Denotational Abstract Interpretation, *Information and Computation* **76** *1* 29–92 (64 sider), 1988.
10. F.Nielson: Towards a Denotational Theory of Abstract Interpretation, *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin (eds.), 219–245 (27 sider), Ellis Horwood, 1987.
11. F.Nielson & H.R.Nielson: Two-Level Semantics and Code Generation, *Theoretical Computer Science* **56** 59–133 (75 sider), 1988.
12. F.Nielson: A Formal Type System for Comparing Partial Evaluators, *Partial Evaluation and Mixed Computation*, D.Bjørner, A.P.Ershov and N.D.Jones (eds.) 349–384 (36 sider), North-Holland, 1988.
13. H.R.Nielson & F.Nielson: Automatic Binding Time Analysis for a typed λ-Calculus, *Science of Computer Programming* **10** 139–176 (38 sider), 1988.
14. F.Nielson & H.R.Nielson: 2-level λ-lifting, *Proceedings CAAP & ESOP 1988*, Springer Lecture Notes in Computer Science **300** 328–343 (16 sider), 1988.
15. F. Nielson: Two-Level Semantics and Abstract Interpretation, *Theoretical Computer Science — Fundamental Studies* **69** 117–242 (126 sider), 1989.
16. H.R.Nielson & F.Nielson: Functional completeness of the mixed λ-calculus and combinatory logic, *Theoretical Computer Science* **70** 99–126 (28 sider), 1990.
17. F.Nielson: The Typed λ-calculus with First-Class Processes, *Proceedings of PARLE 1989*, Springer Lecture Notes in Computer Science **366** 357–373 (17 sider), 1989.
18. H.R.Nielson & F.Nielson: Transformations on higher-order functions, *Functional Programming and Computer Architecture*, ACM Press, 129–143 (15 sider), 1989.

Theoretical Aspects of Semantics-Based Language Implementation

Flemming Nielson
Department of Computer Science
Aarhus University
Ny Munkegade
DK-8000 Aarhus C
Denmark

The research summarised here concerns theoretical aspects involved in the implementation of programming languages directly from a description of their semantics. This involves a study of the subtasks *abstract interpretation* (a framework for program analysis), *code generation* and *program transformation* and the main aim has been to ensure the *correctness* of these subtasks.

1 Introduction

This paper is one of a collection of papers submitted by the author for the Danish degree *doctor scientarum (dr. scient.)* and is written so as to comply with the regulations for that degree. A Danish version of this paper may be found elsewhere.

Section 2 gives a short overview of the subject area of my work. Based on this Section 3 then gives a guided tour of the papers. It stresses the motivation for the individual papers, summarises the main insights gained, explains the main relationships and differences between individual papers and contains comparisons with the literature[1]. It will emerge from this that an initially rather wide outlook has succesively been narrowed so as to obtain results for a class of programming languages rather than just individual toy languages. Section 4 then contains a more detailed discussion of some of the more profound decisions made about the techniques to use and the directions in which to pursue the development. Finally, Section 5 provides additional comparisons with the literature and Section 6 briefly outlines a new aim of my future work.

2 The Subject Area

The general subject area is the development of systems for automating the construction of compilers. One of the main breakthroughs in this area has been the development of parser generator systems that transform context free grammars into parsers for the

[1]Please bear in mind that this paper was written in June of 1989.

language defined. Furthermore, modern LALR(1) based systems produce parsers that are sufficiently efficient to be used in production quality compilers. In a similar way one may hope for systems that transform semantic descriptions, i.e. context free grammars extended with definitions of semantics, into compilers. The fundamental challenge is not only to bring this about but to do so in a way that the resulting compilers may be used as production quality compilers. As should be obvious to anyone who has studied the construction of compilers this challenge is substantially harder to meet than the challenge for parsing.

To assess the quality of a system for developing compilers one will need to consider at least the following ingredients:

- *correctness:* that the compilers produced will generate code that behave as expected, i.e. as the program being compiled,
- *efficiency:* that the code is comparable in efficiency (e.g. time or space) to that of code produced by other means,
- *applicability:* that the programming languages are described in a natural notation (wrt. the kinds of semantics widely used in computer science) and that this notation allows many languages to be defined.

Part of correctness is the task of ensuring that the compilers produced do not loop and that the compiler production system itself does not loop. Related to efficiency is the task of ensuring that the compilers are not too slow and that the compiler construction system itself has a manageable complexity; however, I shall not consider these issues here as they are secondary to the (sufficiently hard) issue of producing efficient code. Finally, applicability of a notation may depend on the class of programming languages under consideration so that e.g. one notation is to be preferred for imperative languages whereas another is to be preferred for applicative (or functional) languages.

To obtain efficient code one may borrow the techniques for program analysis (or data flow analysis) and program transformation (or code optimization) from traditional compiler technology. Correctness then involves

- validating that a (simple-minded) *code generation* scheme is correct,
- validating that the results of the *program analyses* are correct, and
- validating that the *program transformations* preserve the semantics of programs.

The analyses and transformations can be applied both at the source level and at the target level, i.e. one can perform analyses and transformations on the original program and on the code generated for it. As in traditional compiler technology this means that the actual code generation is often kept rather simple-minded because the program may have been transformed so as to facilitate code generation and any shortcomings in the generated code may be alleviated by subsequent transformations. From the point of view of compiler construction one is particularly interested in program transformations that are not meaning-preserving in general but only in certain contexts. Information about the 'contexts' is provided by program analyses and this motivates a close relationship between the program transformations and their enabling program analyses.

3 A Guided Tour

Based on the above description of the subject area this section gives an overview of the papers submitted. These are:

1. F.Nielson: A Denotational Framework for Data Flow Analysis, *Acta Informatica* **18** 265–287 (23 pages), 1982.

2. F.Nielson: Program Transformations in a Denotational Setting, *ACM Transactions on Programming Languages and Systems* **7** *3* 359–379 (21 pages), 1985.

3. A.Mycroft & F.Nielson: Strong Abstract Interpretation using Power Domains, *Proceedings ICALP 1983*, Springer Lecture Notes in Computer Science **154** 536–547 (12 pages), 1983.

4. F.Nielson: Towards Viewing Nondeterminism as Abstract Interpretation, *Foundations of Software Technology & Theoretical Computer Science* **3** (20 pages), 1983.

5. F.Nielson: Abstract Interpretation of Denotational Definitions, *Proceedings STACS 1986*, Springer Lecture Notes in Computer Science **210** 1–20 (20 pages), 1986.

6. F.Nielson: Tensor Products Generalize the Relational Data Flow Analysis Method, *Proceedings 4'th Hungarian Computer Science Conference* 211–225 (15 pages), 1985.

7. F.Nielson: Expected Forms of Data Flow Analysis, *Programs as Data Objects*, Springer Lecture Notes in Computer Science **217** 172–191 (20 pages), 1986.

8. H.R.Nielson & F.Nielson: Semantics Directed Compiling for Functional Languages, *ACM Conference on LISP and Functional Programming* 249–257 (9 pages), 1986.

9. F.Nielson: Strictness Analysis and Denotational Abstract Interpretation, *Information and Computation* **76** *1* 29–92 (64 pages), 1988.

   An extended abstract appeared as [Nie87a].

10. F.Nielson: Towards a Denotational Theory of Abstract Interpretation, *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin (eds.), 219–245 (27 pages), Ellis Horwood, 1987.

11. F.Nielson & H.R.Nielson: Two-Level Semantics and Code Generation, *Theoretical Computer Science* **56** 59–133 (75 pages), 1988.

    Preliminary versions of some of the material of this paper appeared as [NiNi86b, NiNi86c,Nie86b].

12. F.Nielson: A Formal Type System for Comparing Partial Evaluators, *Partial Evaluation and Mixed Computation*, D.Bjørner, A.P.Ershov and N.D.Jones (eds.) 349–384 (36 pages), North-Holland, 1988.

13. H.R.Nielson & F.Nielson: Automatic Binding Time Analysis for a typed λ-Calculus, *Science of Computer Programming* **10** 139–176 (38 pages), 1988.

    An extended abstract appeared as [NiNi88a].

14. F.Nielson & H.R.Nielson: 2-level λ-lifting, *Proceedings CAAP & ESOP 1988*, Springer Lecture Notes in Computer Science **300** 328–343 (16 pages), 1988.

15. F. Nielson: Two-Level Semantics and Abstract Interpretation, *Theoretical Computer Science — Fundamental Studies* **69** 117–242 (126 pages), 1989.

16. H.R.Nielson & F.Nielson: Functional completeness of the mixed λ-calculus and combinatory logic, *Theoretical Computer Science* **70** 99–126 (28 pages), 1990.

17. F.Nielson: The Typed λ-calculus with First-Class Processes, *Proceedings of PARLE 1989*, Springer Lecture Notes in Computer Science **366** 357–373 (17 pages), 1989.

18. H.R.Nielson & F.Nielson: Transformations on higher-order functions, *Functional Programming and Computer Architecture*, ACM Press, 129–143 (15 pages), 1989.

- An English summary (this paper).

- A Danish summary ('Teoretiske aspekter af semantik-baseret sprog-implementation').

Subsections 3.1 and 3.2 give an overview of the papers that deal with 'toy languages', Subsections 3.3 to 3.7 give an overview of the papers that utilise the 'two-level distinction' explained in Subsection 3.3, and Subsection 3.8 considers extending the underlying language. The following table lists for each subsection the papers overviewed there.

3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
1	3	13	5	9	11	12	17
2	4	16	6	15	8	18	
		14	7				
			10				

3.1 Data Flow Analysis and Program Transformation

The motivation for the papers in the first column of the table follows rather directly from the general outlook described in the previous section but restricted to a simple imperative language of `while`-programs. The papers are improvements on, and extensions of, work done in [Nie81] and were motivated by an earlier report [Nie79]. In this I took a rather wide outlook and developed a series of denotational semantics that successively brought a standard semantics into a form where it could be used to generate code for a very simple machine. This development was an attempt at understanding Milne and Strachey's [MiSt76] impressive development of *standard semantics*, *store semantics* and *stack semantics* for an ALGOL-68 like programming language in order to perform its implementation and to prove it correct. From this study it became clear that

- the *optimization aspects*, i.e. the use of program analyses and program transformations, were not yet incorporated in the development of [MiSt76], and

- the development in [MiSt76] was not yet in a form in which one could hope to transform the semantic definitions in an *automatic way* as would be needed in a system for developing compilers.

The first observation motivated [1] in which I formulate a framework of *abstract interpretation* [CoCo79] using a continuation-style *parameterised* denotational semantics for a simple imperative language of `while`-programs. The starting point was a *store semantics* because the 'free variables' or higher-order functions in a *standard semantics* have no analogue in the flow chart model where abstract interpretation and data flow analysis had previously been developed. This does not present a severe limitation as the insights in [MiSt76] may be used to transform a standard semantics into a store semantics. However, the store semantics has no notion of *program point* which is a key notion in the flow chart model and this calls for defining a so-called *collecting semantics* (or *sticky store semantics* in present terminology [10] — see Appendix B) in which program points are introduced and the sets of program states that reach given program points are collected (or *stuck* onto the program points in the sense of glue attracting dust). Only a weak kind of correctness can be expressed between this collecting semantics and the store semantics so the further development is based on an intuitive 'belief' in the similarity of the two semantics (but see below).

The next step then is to formulate a so-called static semantics (or *sticky lifted store semantics* in present terminology) that behaves much as the collecting semantics but on sets of arguments rather than simple arguments. It is precisely characterised in terms of the collecting semantics and is a useful bridge to the induced semantics where the sets of states are replaced by elements of other complete lattices describing more approximate properties. The correctness with respect to the static semantics (and the collecting semantics) is then expressed using the framework of abstract interpretation [CoCo79]. This framework extends the Kam and Ullman approach [KaUl77] to data flow analysis with the use of *adjoined functions* to express the correctness of data flow properties, i.e. to express the sets of states described by the data flow properties.

The main point of [1] is the relationship between the framework for abstract interpretation and traditional data flow analysis. To establish such a relationship I first provide yet another semantics that constructs a traditional flow chart. It is then proved that the *MOP solution* considered in data flow analysis corresponds to the solution specified by the induced semantics. Finally it is shown that the more amenable, but less precise, *MFP solution* considered in data flow analysis can be obtained by modifying the induced semantics (from a continuation-style formulation to a kind of direct-style formulation).

The weak link in the above development is that although the results of the data flow analyses are proved correct with respect to the collecting semantics it is only a matter of 'good belief' that this is also relevant for the store semantics that was our starting point. This link can hardly be improved except by showing that the results obtained may be used to validate program transformations and that the transformed program will have the same meaning as the original program *with respect to the original store semantics.* Note here that there would be no problems if all the program transformations considered would be meaning preserving in general (and thus not depend on data flow analyses).

In [2] two classes of program transformations are considered. One class depends on what is called *forward* [ASU86] and *first-order*[2] data flow analyses. This means that the program is analysed *in the same direction as the flow of control* and that the properties upon which one operates directly *describe data* (e.g. sets of states). For this kind of program transformation it is shown that the collecting semantics can be used to validate program transformations that are not necessarily meaning preserving in general, i.e.

[2]The terminology *first-order* versus *second-order* (see below) was introduced in [2].

whenever certain conditions expressed in terms of the collecting semantics are fulfilled it is proved that the original and the transformed program have the same semantics with respect to the store semantics. For technical reasons I use a collecting semantics where the continuations have been removed [MiSt76].

The other kind of transformations depend on *backward* and *second-order* data flow analyses. This means that the program is analysed *in the opposite direction of the flow of control* and that the properties *do not directly describe data but rather the use of data.* As an analogue of the collecting semantics a so-called *future semantics* is developed that sticks the current continuation onto the program points. With respect to this future semantics it is shown that one can validate the partial correctness of program transformations. Total correctness can then be obtained by (a rather ackward) double transformation. For an example analysis, *live variables analysis*, it is then shown how total correctness may be achieved in a more direct way. — It would seem that few papers in the literature have studied the correctness of program transformations that depend on previous program analyses. A notable exception is [Ger75] but there only partial correctness can be achieved and there are no automatic techniques for inserting the required invariants into every `while`-loop.

3.2 Strong Abstract Interpretation

The above formulation of the collecting semantics was in terms of sets of states. However, the use of powersets, with associated partial order $\subseteq$, does not naturally combine with the use of domains, with associated partial order $\sqsubseteq$. In [1] and [2] this gave rise to the need for considering non-continuous continuations and functions, and in order to ensure that the required least fixed points did exist it was necessary to show seperately that certain functionals were indeed continuous (wrt. $\sqsubseteq$).

The 'nice' solution to these problems is usually claimed to be the replacement of power*sets* by power*domains*. Three well-known powerdomains developed in the 1970'es are the *convex* powerdomain (also called the Plotkin powerdomain after its inventor [Plo76]), the *upper* powerdomain (also called the Smyth powerdomain after its inventor [Smy78]) and the *lower* powerdomain (also called the Hoare or relational powerdomain). The definition of these powerdomains is particularly simple for *flat* domains, i.e. domains with a countable set of elements, one of which is less than the rest but the other elements are incomparable. We write $S_\perp$ for a typical flat domain where the incomparable elements are the elements in the set S and $\perp$ is the least element.

The elements in the upper powerdomain $\mathcal{P}_S(S_\perp)$ are all the finite subsets of S together with the set $S\cup\{\perp\}$ and the partial order is $\supseteq$. As abstract interpretation is concerned with the approximate description of usually infinite sets of values this means that the upper powerdomain is not very useful for our purposes. The lower powerdomain $\mathcal{P}_H(S_\perp)$ consists of all subsets of $S\cup\{\perp\}$ that contain $\perp$ and the partial order is $\subseteq$. This is isomorphic to the powerset $\mathcal{P}(S)$ and turns out to correspond to the development of abstract interpretation as given in [1] and [2]. In this approach the issue of termination is not considered at all, or rather, nontermination is always a possibility.

For certain analyses of programs, notably *strictness analysis*[3] [Myc81], it is important to be able to express that termination is guaranteed rather than always assume that nontermination is a possibility. This means that the lower powerdomain is not suitable for all aspects of strictness analysis and so one might consider the convex powerdomain

[3]The associated program transformation is the transformation of call-by-name into call-by-value.

$\mathcal{P}_P(S_\perp)$. The elements are the nonempty subsets of $S_\perp$ but there is the constraint that they must be finite if they do not contain $\perp$. The partial order is the so-called Egli-Milner order $\sqsubseteq_{EM}$. However, the restriction about finiteness means that one cannot describe a set like S that describes *all* possible outcomes of a computation that is guaranteed to terminate. As a consequence one of the analyses of [Myc81] could not be validated.

This motivated the development of a powerdomain $\mathcal{P}_N(S_\perp)$ in which a set like S could be described. In [3] such a powerdomain is defined in the context of a simple applicative language of recursion equation schemes. The powerdomain $\mathcal{P}_N(S_\perp)$ has all nonempty subsets of $S_\perp$ as elements and the partial order is the Egli-Milner order $\sqsubseteq_{EM}$. (General conditions for the existence of the present powerdomain are given in [4].) The price to pay for having infinite sets without $\perp$ is that certain extension functions associated with the powerdomain are monotonic but not necessarily continuous. Therefore the domains used must have least upper bounds of all directed sets and to ensure the existence of fixed points of a monotonic function we need to iterate the function beyond the ordinal ω. It follows from the potential non-continuity of the extension functions that this powerdomain differs from the generalised powerdomain developed in [Plo82] as well as from the three powerdomains mentioned above.

The Egli-Milner order is not the appropriate order to use for comparing sets of arguments from the point of view of abstract interpretation. This motivates the consideration of *augmented domains*, i.e. domains with an additional partial order (that satisfies certian conditions) and in case of the powerdomains this additional partial order is subset inclusion $\subseteq$. This is close in spirit to the development of the so-called nondeterministic cpo's used in [HeAs80] to give a denotational semantics for nondeterministic programs.

The *novel ingredient* is that abstract interpretation is developed for pairs of abstraction and concretization functions that are monotonic with respect to both partial orders but adjoined only with respect to the augmented partial order. Additionally we need the concretization functions to satisfy conditions called pseudo-strictness and pseudo-continuity. In this setting we can establish the correctness of analyses and prove that the induced analyses are as precise as possible.

It is then shown how to validate the two strictness analyses of Mycroft [Myc81] and this involves interpreting the augmented partial order in various ways. When interpreting it as the usual Scott-order, $\sqsubseteq$, one obtains a theory of abstract interpretation much as in [1] and [2]. — The important insight gained by this development is that in general the Scott-partial order, $\sqsubseteq$, used to ensure well-definedness of denotational semantics is fundamentally different from the partial order, $\subseteq$, used to compare properties in abstract interpretation. This point was already made in [1] but on more intuitive grounds.

Motivated by the similarity between the mathematical tools used to describe nondeterminism and abstract interpretation one may study whether nondeterminism may be explained as abstract interpretation. This is shown to be the case in [4] and the basic motivation is straightforward: Usually nondeterminism is motivated by the need to abstract away from certain physical or implementation oriented aspects of programs running on a computer system. But with perfect knowledge one might well be able to explain the precise outcome and this approach is indeed taken in early papers (e.g. [Mil75]).

For a simple language of `while`-programs I assume the existence of *oracles*, i.e. an infinite sequences of 0's and 1's, that can be used to resolve each 'nondeterministic choice'. Then [4] performs a modified development of abstract interpretation where only some of the components in the arguments to functions are replaced by properties or sets of values. For the example language the sets of oracles are replaced by a property that describes all oracles, i.e. all infinite sequences of 0's and 1's. It is shown that the resulting abstract interpretation corresponds to the usual nondeterministic semantics defined for such a language. — Thus with a sufficiently powerful concept of 'powerdomain', nondeterminism may be regarded as an application of abstract interpretation.

Going further into the nature of oracles it is shown that one may restrict one's attention to a countable set of oracles but that such a set would need to contain the recursive oracles as a *proper* subset. This shows that if the 'physical or implementation oriented' phenomena that one abstracts away from are indeed computable then the nondeterministic semantics describes divergence too often. The problem only arises for the iterative constructs, i.e. for least fixed points, and [4] concludes by identifying another fixed point operator that specifies divergence less frequently. It is shown that this fixed point operator is the top element in a complete lattice of 'acceptable fixed point operators' and that the least fixed point operator is the bottom element. It is furthermore shown how an operating system could enforce the semantics specified by the newly exhibited fixed point operator.

3.3 TML: Compile-Time versus Run-Time

The two approaches to abstract interpretation overviewed above do not provide the degree of *automation* needed for use in a compiler development tool. And thus they do not successfully address both of the observations made in the first paragraph of Subsection 3.1. However, the insights obtained constitute a useful platform for trying to distill the ingredients needed for a higher degree of automation.

Such a development *has to* focus on a *metalanguage* for (denotational[4]) semantics as the compiler development tool must accept a semantics and process it automatically. Thus the role of the *metalanguage* is quite analogous to the role of BNF in the specification of a programming language (e.g. as input to a parser generator). Usually a denotational metalanguage is based on the typed λ-calculus and we shall assume that the metalanguage is little more than this (except when we come to Subsection 3.8 below). The main shortcoming of this metalanguage is that it does not give sufficient flexibility in how to interpret various constructs. This is clearly demonstrated by the recursion equation schemes of [3]. Consider a program of the form

```
let f(x,y) = ...
and g(x,y) = ...
in f
```

where x is an integer argument, y a boolean argument and the results of the functions are integers. The semantics of this program will be of the form

$$\mathrm{FIX}(\ F\ ) \downarrow 1$$

for some semantic functional F. The details of F are of no concern here but merely its functionality. When defining the *standard semantics*, i.e. the input-output meaning of programs, one might have

$$F \in (\mathbf{Z}\times\mathbf{B}\to\mathbf{Z}) \times (\mathbf{Z}\times\mathbf{B}\to\mathbf{Z}) \to (\mathbf{Z}\times\mathbf{B}\to\mathbf{Z}) \times (\mathbf{Z}\times\mathbf{B}\to\mathbf{Z})$$

[4]The choice of denotational semantics is discussed in Section 4.

where $\mathbf{Z}$ is the flat domain of integers and $\mathbf{B}$ is the flat domain of booleans. Next we turn to a few program analyses. The details of these are of no concern here and we shall not give detailed explanations (but refer the uninitiated reader to [ASU86]). In a constant propagation analysis one might have

$$F \in (\mathbf{L}\times\mathbf{M}\to\mathbf{L}) \times (\mathbf{L}\times\mathbf{M}\to\mathbf{L}) \to (\mathbf{L}\times\mathbf{M}\to\mathbf{L}) \times (\mathbf{L}\times\mathbf{M}\to\mathbf{L})$$

where $\mathbf{L}$ is a complete lattice ($\mathbf{Z}$ with a top element added) and $\mathbf{M}$ is a complete lattice ($\mathbf{B}$ with a top element added). This analysis is an *independent attribute analysis* because no relationship between the integer and boolean arguments are taken into account. If formulated as a *relational analysis* one could express relationships between the integer and boolean arguments and the functionality would then be

$$F \in (\mathbf{L}\otimes\mathbf{M}\to\mathbf{L}) \times (\mathbf{L}\otimes\mathbf{M}\to\mathbf{L}) \to (\mathbf{L}\otimes\mathbf{M}\to\mathbf{L}) \times (\mathbf{L}\otimes\mathbf{M}\to\mathbf{L})$$

for a suitable operator $\otimes$. Both of these analyses are *forward analyses* [ASU86]. Turning to a *backward analysis* like live variables analysis the functionality might be

$$F \in (\mathrm{N}\times\mathrm{N}\leftarrow\mathrm{N}) \times (\mathrm{N}\times\mathrm{N}\leftarrow\mathrm{N}) \to (\mathrm{N}\times\mathrm{N}\leftarrow\mathrm{N}) \times (\mathrm{N}\times\mathrm{N}\leftarrow\mathrm{N})$$

where N is a complete lattice, e.g. with elements `live` and `dead`, and $\mathrm{N}\times\mathrm{N}\leftarrow\mathrm{N}$ is a shorthand for $\mathrm{N}\to\mathrm{N}\times\mathrm{N}$.

What is important in the above overview of semantics and analyses is to note the way in which the functionality of F changes. The various base types, like the integers, may be interpreted by different domains and complete lattices, e.g. $\mathbf{Z}$, $\mathbf{L}$ and $\mathbf{N}$. Also the type constructors need to be interpreted differently depending on whether the analysis is in independent attribute form or in relational form and whether it is forward or backward. However, we should also note that some constructors remain fixed throughout. These are the function space construction $\to$ used to describe the computation that F performs and the cartesian products $\times$ that describe that F takes an argument that is a pair (of the meanings of `f` and `g`) and produces a result that is a pair (of the new meanings of `f` and `g`).

We thus need a version of the typed λ-calculus in which F can be assigned a type that is flexible enough for all of the above types to be obtainable by suitable interpretations or instantiations. One possibility therefore is to use

$$F \in \underline{\mathrm{Int}\times\mathrm{Bool}\to\mathrm{Int}} \times \underline{\mathrm{Int}\times\mathrm{Bool}\to\mathrm{Int}} \to \underline{\mathrm{Int}\times\mathrm{Bool}\to\mathrm{Int}} \times \underline{\mathrm{Int}\times\mathrm{Bool}\to\mathrm{Int}}$$

The *primary purpose* of underlining is to be able to distinguish between types and type constructors that should be interpreted or instantiated differently. However, it also emerges that the non-underlined (types and) type constructors may be interpreted in the same way regardless of the analysis being performed. A potentially more flexible type would be

$$F \in \underline{\mathrm{Int}\times\mathrm{Bool}\to\mathrm{Int}} \times \underline{\underline{\mathrm{Int}\times\mathrm{Bool}\to\mathrm{Int}}} \to \underline{\mathrm{Int}\times\mathrm{Bool}\to\mathrm{Int}} \times \underline{\underline{\mathrm{Int}\times\mathrm{Bool}\to\mathrm{Int}}}$$

since it e.g. allows different interpretations of $\underline{\mathrm{Int}}$ and $\underline{\underline{\mathrm{Int}}}$. From a theoretical point of view this hardly makes any difference so to cut down on the notational complexity the simpler type will have to suffice.

The foundation for the metalanguage will be a type scheme as indicated above. Clearly the distinction on types that the underlining gives rise to necessitates a similar distinction on expressions. We may define a *two-level λ-calculus* to have types as above and expressions that are as in the λ-calculus except that the constructs may be underlined. (This is of course subject to well-formedness constraints and these may be found in the papers.) However, the two-level λ-calculus does not make it sufficiently easy to interpret the 'underlined' constructs as freely as needed. In analogy with the focus on 'categorical combinators' for interpreting the typed λ-calculus in a cartesian closed category this motivates a notation where the 'underlined' λ-constructs are replaced by combinators. (There is no need to make a similar change for the 'non-underlined' λ-constructs as they always mean the same.) The resulting notation is called TML, for *two-level metalanguage*, and is the metalanguage upon which most of the subsequent development is based. As with the two-level λ-calculus also TML is subject to certain well-formedness constraints. The details of these vary but there is a general trend to make the constraints less demanding (and to enrich the metalanguage) and the numbering of the papers roughly indicates the order in which their 'essential structure' was fixed.

The well-formedness constraints imposed are partly of a technical nature and partly a formalisation of the distinction between compile-time and run-time. (An exception is [JoNi91, Section 3] where I develop a more liberal scheme.) The basic idea is that the underlined constructs, whether types or expressions, correspond to run-time values or computations, whereas the non-underlined constructs correspond to compile-time values or computations. An example constraint is that a run-time type cannot involve a compile-time type as run-time only takes place after compile-time. Precursors of a formal distinction between compile-time and run-time are mentioned in some of the papers; here we only note Tennent's [Ten81] informal distinction between *static expression procedures* and *expression procedures* and Paulson's [Pau84] *semantic grammars.* — Since compile-time and run-time are fundamental concepts in compiler construction one may regard the use of two-level types and expressions as a way of increasing the fit between (informal) *computer science concepts* and a widely studied (formal) *model* of types and λ-expressions.

The intuitions behind 'compile-time' and 'run-time' may be used as an aid in introducing underlining into an otherwise non-underlined λ-expression (as is discussed in Subsection 3.6 below). Proceeding along this line of thought one might consider whether underlining may be introduced in a mostly automatic way. Starting with the two-level λ-calculus it is shown in [13] that there exists an algorithm that given

> an ordinary typed λ-expression and its overall type augmented by underlining *some* of the constructs or types that need to be deferred to run-time

will return

> a *best two-level λ-expression*, i.e. one that satisfies the well-formedness constraints and where as few computations as possible are deferred to run-time.

(This is all formalised by the definition of suitable partial orders.) This *binding time analysis algorithm* is motivated by R.Milner's algorithm $\mathcal{W}$ [Mil78] for polymorphic type inference. However, the present algorithm for binding time analysis uses a more complicated structure of recursive calls as [13] has no analogue of the composition of type substitutions used in [Mil78]. The proof of correctness therefore is not by structural induction but by induction on a more complex wellfounded order.

Turning to the combinators it is natural to adapt the techniques for combinator introduction, or bracket abstraction, (e.g. [Sch24], [CuFe58], [Tur79] and [Hug82]) to the two-level case. It turns out that the two-level structure does pose some additional complications and [14] develops a transformation, called *two-level λ-lifting*, that can handle a subset of the two-level λ-calculus. The full two-level λ-calculus is handled in [16] but at the expense of rather involved transformations and the need to consider a combinator, Ψ, that may be regarded as 'un-implementable' [NiNi89]. From a pragmatic point of view it therefore seems better to restrict oneself to a subset of the two-level λ-calculus. This subset may be that of [14] although it would be desirable to extend the subset using the (incomparable) techniques developed in [16] for avoiding Ψ but to do so still presents an open problem. It is straightforward, however, to adapt the binding time analysis so that it always produces two-level expressions in the subset considered in [14] (as in [NiNi90, Chapter 3]).

A rather informal overview of binding time analysis, two-level λ-lifting and much of the subsequent development may be found in [NiNi88b]. In particular, [NiNi88b] focuses on how to combine the various subtasks that are treated in more isolation in the other papers overviewed in Subsections 3.3 to 3.7.

3.4 Abstract Interpretation: Collecting Semantics Based

A series of papers develop abstract interpretation for a subset, TMLs, of the metalanguage in which nesting of underlined function types is disallowed. In practical terms this means that the development is restricted to PASCAL-like languages, rather than general functional languages, as one cannot model storable procedures. An additional constraint needed for parts of the technical development (and also used in [16] for avoiding Ψ) is that the types of certain base functions must satisfy a constraint called *'contravariantly pure'*. This constraint is easiest to explain in terms of code generation (as is overviewed in Subsection 3.6) and then says that when a compiler generates code it is not allowed to run a piece of code and then use the performance of this as an aid in determining what code to generate.

A terse presentation of the development is given in [5] and is based on [Nie84]. The semantics of types (including recursive types at the run-time level as well as at the compile-time level) is formulated using the category-theoretic approach to domain theory [SmPl82]. Concerning run-time types we shall see below that the development needs to be able to handle the tensor product which, unfortunately, is not a functor over the category of algebraic lattices (i.e. domains that are complete lattices) and strict continuous functions. Rather than restricting the attention to the subcategory of additive functions, the development in [5] modifies the definition of functor to a weaker concept called semi-functor. (It turns out that such definitions are not completely estranged to the category-theorist although they fall outside the usual textbook treatments of category theory.) Concerning the compile-time types the contravariance of function space is troublesome and rather than restricting the attention to the subcategory of embeddings (or lower adjoints) the development in [5] follows [Rey74] in using a category of domains where morphisms are pairs of continuous functions of the form $(f{:}D\rightarrow E, g{:}E\rightarrow D)$. Over this category the function space constructor becomes a covariant functor (as the 'contravariance' is hidden in the definition of composition in the category).

In this framework an eager standard semantics for TMLs is defined. (Eagerness corresponds to the use of strict functions, smash product, coalesced sum etc.) Next a version of the collecting semantics (or *lifted standard semantics*[5] in present terminology)

[5]We use the term standard semantics although the constraint on no nested run-time function types

is defined but *without* the notion of program points, i.e. the only two program points considered are the program-entrance and the program-exit. The formulation uses the lower powerdomain discussed above and it is shown that the semantics of a run-time type in the collecting semantics is (isomorphic to) the powerdomain of the semantics of the same type in the standard semantics. The proof is by induction on run-time types and due to the presence of type variables the induction hypothesis is slightly stronger and is expressed in terms of natural equivalences. This result is the technical motivation for why nested run-time function types have been disallowed because it seems impossible to find an operator $\boxed{\rightarrow}$ such that $\mathcal{P}_{\mathrm{H}}(D) \boxed{\rightarrow} \mathcal{P}_{\mathrm{H}}(E)$ is isomorphic to $\mathcal{P}_{\mathrm{H}}(D\rightarrow E)$, where $\rightarrow$ denotes strict function space. Furthermore, this result motivates the need to consider tensor products as the tensor product $\otimes$ satisfies that $\mathcal{P}_{\mathrm{H}}(D)\otimes\mathcal{P}_{\mathrm{H}}(E)$ is isomorphic to $\mathcal{P}_{\mathrm{H}}(D\star E)$, where $\star$ denotes smash product.

The semantics of expressions amounts to a structural definition of elements in the required domains. In [5] the standard semantics and the collecting semantics are defined for expressions. To show the correctness, or precision, of the collecting semantics we need relations that relate values in the standard semantics to values in the collecting semantics. These relations are defined by induction on compile-time types using the notion of logical relations [Plo80] or relational functors [Rey74]. The required connection is then proved using a lemma stating that the semantics of types and the definition of the relations together constitute a definition of a functor over a suitable category.

Abstract interpretation is then developed. The first step is to formulate a scheme for the definition of adjoined pairs $(\alpha_{\mathrm{rt}},\gamma_{\mathrm{rt}})$ by induction on the structure of run-time types (rt). For the run-time type constructors this makes use of the notion of natural transformations but suitably adapted to semi-functors rather than functors. In practical terms the natural transformations formalise how to shift between various formulations of analyses, e.g. from relational form to independent attribute form. The next step then is to link up with the relations exhibited above and show that if the primitives of TMLs are analysed correctly then also every expression in TMLs is.

Having ensured the correctness of analyses one may consider whether 'most precise analyses' (or *best induced predicate transformers* [CoCo79]) exist over a given selection of properties. Using the restriction that basic functions must have contravariantly pure types it is shown that this always is possible and essentially amounts to extending the definition of the adjoined pairs $(\alpha_{\mathrm{rt}},\gamma_{\mathrm{rt}})$ to adjoined pairs $(\alpha_{\mathrm{ct}},\gamma_{\mathrm{ct}})$ ranged over by compile-time types (ct). To be more specific, given an analysis (e.g. the collecting semantics that has already been related to the standard semantics) and given a selection of properties one can construct an *induced* analysis over those properties. The induced analysis is correct and is approximated by any other correct analysis over the same selection of properties. As the definition of $(\alpha_{\mathrm{ct}},\gamma_{\mathrm{ct}})$ turns out to behave in a functorial way (wrt. the 'arguments' α and γ) one can develop an induced analysis in many small steps and obtain the same result as if the development had been performed in one big step.

The interpretation of run-time types (rt) is compositional and hence the interpretation of the underlined type constructors are of particular interest. Taking underlined product as an example one natural interpretation is as cartesian product and as illustrated above this corresponds to an analysis in independent attribute form. Another possibility is to use a tensor product and given the characterisation of the tensor product on powerdomains this shows that, at least on powerdomains, the tensor product corresponds to an

essentially restricts the semantics to be in the form of a store semantics as in [1] and [2].

analysis in relational form. However, the notion of 'relational form' is inherently an intuitive notion. To validate the claim that 'tensor product' formalises 'relational form' in general, the development in [6] characterises the tensor product on a class of complete lattices that includes the powerdomains as a special case and in this way substantiates the general claim. Also [6] performs a systematic study of how to *shift* between various data flow analysis formulations (using natural transformations between semi-functors). — The insight to be gained from this kind of study is that important distinctions in the formulation of data flow analyses may be formalised by the way (run-time) type constructors are interpreted.

The existence of the induced analysis is very pleasant from a theoretical point of view but an induced analysis is not necessarily computable just as the collecting semantics is not necessarily computable. So for reasons of computability, and for other pragmatic reasons, one may consider some of the key primitives in TMLs and suggest certain *expected forms* for their definition. These expected forms should be reasonably straightforward to implement and there should be general results stating that if one uses expected forms throughout then the correctness of shifting between analyses can be guaranteed independently of the actual analyses. Such a development is performed in [7] where several interpretations of the run-time type constructors are considered and for each interpretation expected forms are suggested. It is then proved, by case analysis on the various combinations, that the correctness of the expected forms may be guaranteed in general.

An overview of this development as well as a more general motivation is given in [10]. This paper also goes into a discussion of the suitability of powerdomains as the convexity that is present in all (known) powerdomains pose certain problems for applications. Finally, it outlines in what way a development using program points would relate to the development conducted here. (Elements of the presentation build on the insights obtained in [9] below.)

3.5 Abstract Interpretation: Standard Semantics Based

The subset TMLs of the two-level metalanguage excludes nested run-time function types, i.e. storable procedures, and so is less general than one would have hoped for. The main motivation for this restriction was in order for the collecting semantics to be defined. Together with the development in [BHA86] this motivates modifying the above development so as to be independent of the existence of the collecting semantics.

A first step in this direction is taken in [9]. Here I consider a small subset TMLb of the two-level metalanguage where no compile-time recursive types are allowed and where the structure of the run-time types is left completely unspecified. Thus TMLb may be regarded as a subset of TMLs (ignoring a few minor differences in formulation). The absence of recursive types is motivated by certain technical problems mentioned later. The completely unspecified nature of the run-time types is partly motivated by the choice of language in [BHA86] and partly by the fact that the difficulties in formulating the collecting semantics above only occurred for the underlined run-time type constructors.

The definitions of syntax and parameterised semantics given in [9] are along the same lines as above. However, *correctness* of an analysis is now formulated as a relation to the standard semantics rather than as a relation to the collecting semantics thus not relying on the existence of the collecting semantics. This, of course, specialises to the correctness of one analysis with respect to another analysis (e.g. the collecting semantics, if it exists) and due to the special nature of this relation the development in [9] uses the term *safety* for this kind of correctness. Having formulated correctness the next step is to ensure the existence of *induced analyses*. These can be specified from other analyses much as above but since we only assume the existence of the standard semantics, and not of the collecting semantics, this does not allow us to get the 'inducing of analyses' started.

Based on the equations for inducing an analysis from another analysis one can formulate equations for inducing an analysis from the standard semantics. However, the equations provided turn out not to be well-defined in general. This leads to a notion of *faithful sets* of elements. One of the important consequences is that the relation $\leq$, an extension of $\sqsubseteq$ indexed by two-level types, behaves as a partial order on faithful elements. It can then be shown that the equations for inducing are well-defined and behave as expected, if one restricts the attention to faithful elements. Continuing this development one can show that analyses may be developed by a process of 'stepwise coarsening', i.e. rather than inducing an analysis directly from the standard semantics one may induce it from other analyses that have already been induced from the standard semantics and the resulting analyses will then be the same.

The need to restrict oneself to the subset of faithful elements in a domain makes the development somewhat complicated. This leads to a syntactic constraint, *level-preserving*, on types and this constraint is sufficient to ensure that all elements in the semantics of the types are faithful. Any contravariantly pure type is also level-preserving so the present development extends parts of the development in the previous subsection. It is also shown that the approaches of [MyJo86] and [BHA86] may be handled in the present approach. — As [BHA86] only considers strictness analysis and [MyJo86] does not consider inducing it seems fair to claim that the present approach provides better insight into the relative precision of analyses including strictness analysis as a special case.

A main motivation behind [9] was to avoid basing the development on the existence of the collecting semantics as this posed problems in certains contexts (see above) and had been questioned by others (e.g. [MyJo86]). This motivates a study of whether the collecting semantics exists for TMLB and whether correctness and inducing wrt. the standard semantics is equivalent to correctness and inducing wrt. the collecting semantics. These questions are all answered in the affirmative in [9]. (With respect to the problems of the previous subsection recall that the nature of the run-time types are left unspecified in TMLB.)

To extend the applicability of this development the rather small subset TMLB of the two-level metalanguage must be augmented. However, already recursive types at the compile-time level pose problems for the development in [9] as functions only are assumed to be monotonic (as is discussed in the Conclusion of [9]). Also the nature of the run-time types must be specified in a structural way.

This motivates the development in [15] that combines the generality of the previous subsection with the above insights about not assuming the existence of the collecting semantics. The development studies a metalanguage TMLM that lifts the constraint in TMLS about not having nested run-time function types. So TMLM has base types, product types, function types, sum types and recursive types at the compile-time level as well as at the run-time level. In this way the metalanguage should be applicable to general functional languages rather than only PASCAL-like languages. Furthermore, the preferred approach to two-level λ-lifting (i.e. [14]) produces expressions in (a slightly modified version of) TMLM.

The main ideas behind the development in [15] are:

- to concentrate on *level-preserving* types and to disallow primitives to have types that are not level-preserving,

- to formulate the equations for inducing in terms of the Scott-order, $\sqsubseteq$, rather than the partial order motivated by abstract interpretation, $\leq$, and
- once well-definedness of these equations has been shown, to develop equivalent equations in terms of $\leq$, i.e. to demonstrate the required properties of the equations.

To be more specific, Section 2 of [15] presents an overview of abstract interpretation and discusses the approaches taken by different authors and how the approach and results of [15] relate to these. The actual metalanguage, TMLm, underlying the study is defined and motivated in Section 3 and Section 4 develops the notion of parameterised semantics. For compile-time types this follows the overall theme of [5]. For run-time types added generality wrt. [5] is obtained by allowing nesting of run-time function types and by allowing the semantics of run-time function and recursive types to depend on the interpretation. This calls for a somewhat more general setup than in [5] where now also the run-time types are interpreted over a category where morphisms are pairs of continuous functions of the form $(f{:}D{\rightarrow}E, g{:}E{\rightarrow}D)$. This notion of semantics is illustrated by defining two standard semantics (one eager and one lazy) and two analyses (detection of signs and liveness).

Section 5 is devoted to the formulation and demonstration of correctness of an analysis with respect to the standard semantics (or another analysis). This is much as surveyed in the previous subsection except that it is also indicated how the correctness relations may be defined structurally on run-time types. This is then used to demonstrate the correctness of the detection of signs analysis and of the liveness analysis (using ideas from [2]) and the latter was not possible in [5].

Sections 6 and 7 study the construction of induced analyses and concentrate on the compile-time types and the run-time types, respectively. Both sections begin with a study of the (easier) adjoined case where one analysis is specified in terms of another, then go on to study the general case where an analysis is specified in terms of a standard semantics and finally discuss the existence of and the role of the collecting semantics.

For the study of the compile-time types the structure of the run-time types is ignored and thus it is not assumed that the semantics of the run-time types is specified in a compositional manner. This brings the development rather close to that in [9] but a major difference is that inducing only is considered for level-preserving types. An advantage of the restriction to level-preserving types is that it becomes rather straightforward to characterise the compact elements in the domains; hence one can work with continuous functions (rather than just monotonic functions) so that also recursive types may be handled. Another advantage is that the proof of the properties of inducing becomes simpler because one can exploit the definition of 'level-preserving' in a case analysis. Using the insights of [9] it is proved that best induced analyses may be constructed from a standard semantics and that analyses may be constructed by a process of stepwise coarsening (along the lines of [9] above). As would be expected from [9] it is also shown that the collecting semantics exists in general (recalling that the structure of the run-time types is *not* taken into account).

For the study of the run-time types natural transformations are used to shift between the interpretations of the various type constructors and a notion called 'natural adjunction transformer' is used to shift between interpretations of the run-time recursive types. (A proper categorical concept eluded me here due to the need to pass from a functor over one category to a functor over another category.) This setup is illustrated with examples of how to relate the interpretation of various base types, of various type constructors (including a shift from relational form to independent attribute form) and of approximating recursive types by finite unfoldings. Finally, there are problems similar to those in [5] when trying to demonstrate the existence of the collecting semantics (recalling that the structure of the run-time types *is* taken into account). — Hence the present development is more satisfactory than [5] in that the existence of the collecting semantics no longer is a prerequisite to the development of abstract interpretation.

Section 8 then studies expected forms much as in [7] but generalised so as to handle the extended metalanguage. Section 9 concludes by summarising the main insights gained and a total of four appendices provide proofs not supplied in the main text.

3.6 Code Generation

Program analyses by means of abstract interpretation (or any other means) are not the only ingredient in the construction of compilers. An even more central ingredient is the actual code generation and it would therefore be desirable if the two-level metalanguage would be a suitable platform for a general theory of code generation just as it is a suitable platform for a general theory of abstract interpretation. In fact this potential application of the two-level metalanguage has guided its definition all along (as may be seen from a remark in [Nie84]) and it is also interesting to observe that when the desire to perform code generation is abandoned one can allow a more liberal type structure and still develop a theory of abstract interpretation (as I do in [JoNi91, Section 3]).

A theory of code generation for the subset TMLs is developed in [11]. Section 2 motivates the metalanguage and Section 3 defines the notion of parameterised semantics much as in the papers on abstract interpretation. The standard semantics is an eager standard semantics, i.e. it uses strict function space, smash product and coalesced sum.

Section 4 then defines a simple abstract machine called AM. To be realistic, at least for the purposes of implementing functional languages, it is a simplified version of the Functional Abstract Machine, FAM, used by L. Cardelli [Car83] in his implementation of the programming language ML. As already FAM is a natural machine for the implementation of functional languages (including the λ-calculus) it should not be surprising that AM has many similarities with the Categorical Abstract Machine, CAM, of [CCM87]. However, AM uses symbolic labels in the code and is closer to a traditional machine in that code is a linear sequence of instructions.

The semantics of the machine is formalised by a transition system that operates on configurations consisting of a program counter, a stack of values and a stack of return addresses. Each inference in the transition system is intended to model a single step of a machine. The meanings of the program counter and stack of return addresses should be rather clear. The meaning of the elements on the stack of values is less clear, however, as the standard semantics works on values in semantic domains whereas the transition system works on values (in the configuration) that are essentially just 'bit-patterns'. To make the connection between these two kinds of values clear, and to prepare for the correctness proof, *representation functions* are defined for mapping the run-time values to 'bit-patterns'. Elements of recursive types are represented by their unfoldings (which, due to the eagerness of the standard semantics, necessarily are finite).

Section 5 defines an interpretation that generates code for the abstract machine. This is mostly straightforward and only the case of fixed points requires special techniques.

Here the fixed point of a run-time function type should give rise to the usual kind of code where the recursive call is performed by a *call*-instruction that pushes a return address on the stack of return addresses. Mutual recursion may be handled using Bekic's theorem (for reducing mutual recursion to nested single recursion). A similar trick can be performed for sums of types. For a compile-time type that contains no run-time types no code should be generated and so one may use the usual least fixed point constructor in this case. (From a pragmatic point of view one should verify that the compiler cannot loop but at present I have no techniques for detecting when this might occur.) As discussed in [11] there are problems in handling other types than those considered so far and this motivates defining a syntactic constraint called *composite* and only allow fixed points of composite types.

Section 6 studies how to live with the constraints imposed above, i.e. the need to consider only TMLS and the need to constrain certain types to be composite. It is shown that a semantics for the language SMALL [Gor79] may be systematically transformed into the required subset of the two-level metalanguage using the techniques of continuation-removal and replacement of a (dynamic) location by a (static) pair of block number and block offset. The last technique is of particular interest for two reasons. One is that it also occurs in [MiSt76] as part of the development of a *stack semantics* from a *standard semantics* but unlike the development in [MiSt76] the application here is clearly motivated (as a way of passing from one subset of TML to a smaller one). The second reason is that it shows how the techniques and tricks from compiler construction may enter into the area of *semantics-directed compiler construction*: Initially one has a semantics in a rather large subset of the two-level metalanguage but efficient implementation techniques are only developed for certain smaller subsets and so one uses heuristics for transforming a semantics into smaller and smaller subsets. — I believe this 'link' between traditional compiler technology and semantics directed compiling to be new.

Section 7 (and the appendix) then demonstrates correctness. This is done in four stages. The first stage amounts to defining the correctness relation for run-time function types and this is accomplished using the representation functions defined previously. It also defines auxiliary relations needed to express that the code is well-behaved, e.g. so that one can insert dummy instructions without changing the semantics of the code. In the second stage these relations are extended to all compile-time types. The basic idea is to use the framework of logical relations as was done for abstract interpretation. However, the strategy used in the correctness proof calls for first proving that the code is well-behaved and then to use this information when proving actual correctness. It seems unclear how to express this using logical relations and the stronger framework of Kripke-like relations is therefore used for defining the required relations at higher-order compile-time types. (Essentially, a Kripke-like relation is a logical relation that takes an additional 'parameter' from a partially ordered set.) Turning to the correctness proof the third stage proves that the primitives are handled correctly. This is mostly straightforward (and exceedingly tedious) but in the case of fixed points the proof considers a modified machine that can be made to loop when a certain level of recursion is encountered. (Information about the level of recursion is part of the 'parameter' supplied to the Kripke-like relations.) In the fourth and final stage this is then used as a basis for a proof (by structural induction on expressions) that correct code is generated for all expressions in the designated subset of the two-level metalanguage.

Ideally the approach outlined above should be applicable to the subset TMLM for which abstract interpretation was developed above. Concerning the formulation of an interpretation for code generation this turns out not to present any problems and [8] defines coding interpretations corresponding to eager semantics as well as lazy semantics. Concerning the proof of correctness there are serious problems as even the formulation of correctness does not easily generalise from [11]. (The problem is that one cannot expect a reasonably well-behaved representation function to exist for run-time function types as allowed in TMLm but not TMLs.) Thus [8] does not prove correctness and to overcome this one might look into the techniques used in [Plo77].

3.7 Transformations on Programs

We have already seen the use of program transformations above. In Subsection 3.1 I demonstrated the use of data flow information stuck onto program points (of a simple language of `while`-programs) for various program transformations. In Subsection 3.6 I demonstrated how continuation removal and the representation of (dynamic) locations as (static) pairs of block number and block offset might be used to obtain an implementation. Also the binding time analyses and combinator introduction algorithms of Subsection 3.3 might be thought of as a kind of program transformation.

In [12] a bridge is built between the two-level type structure and a class of program transformations performed by *partial evaluation.* Partial evaluation is modelled by an operational semantics, i.e. a reduction relation involving at the very least β-reduction, but constrained so that *only occurrences of compile-time constructs may be rewritten.* In practical terms one would expect this rewriting relation always to terminate because a compiler (and hence a two-level semantics) is usually constructed so that it cannot loop but as already discussed above techniques are needed for enforcing this.

Unfortunately the workshop at which [12] was presented seemed to have no consensus on what is permissible as partial evaluation. It would seem that most researchers would accept the above calculations as *part* of partial evaluation but that some would allow *more* powerful techniques, e.g. program transformations using techniques from theorem proving. Given the lack of consensus on what constitutes partial evaluation there necessarily must be a lack of consensus on when some partial evaluator is *better* than another. However, given a perspective on what constitutes partial evaluation one can study ways of formalising the relative merits of various partial evaluators. The approach in [12] was motivated by the lack of formality[6] in many papers on partial evaluation and by the belief that firm arguments cannot be provided before some definitions have been presented. This is analogous to abstract interpretation where the correctness of an analysis could be argued only on intuitive grounds as long as there was no precise definition of correctness but once such a definition was given (by defining relations like $\leq$) more convincing arguments could be given.

So with the perspective on partial evaluation enunciated above, [12] goes on to propose three relations that might formalise the notion of 'being a better partial evaluator than' and the relative merits of these definitions are discussed.

Note in passing that [12] contains the first presentation of the two-level λ-calculus (as opposed to the two-level metalanguage) and contains the first characterisation of binding time analysis for a typed λ-calculus. Furthermore, the conclusion discusses how certain restrictions in the two-level type structure (that are helpful for code generation[7]) could be lifted so as to fit more closely with other approaches to partial evaluation (e.g.

[6]Thus tacitly assuming that formality is a virtue.

[7]One should note that the $\Downarrow$ in [12] is closely related to the combinator Ψ 'rejected' in Subsection 3.3.

[JSS85]). (This is in analogy with, but not similar to, the liberalisations of the two-level metalanguage performed in [JoNi91, Section 3].)

A much broader perspective on the use of program transformations is presented in [18]. The development centers on a simple example and does not attempt to guarantee that the transformations can be applied in an *automatic way*. This approach allows for a more coherent presentation of the way in which the current research is hoped to interact with program transformations.

The program transformations illustrated in [18] are roughly divided into three groups. One group of transformations is intended to aid the binding time analysis. The binding time analysis algorithm will always succeed in producing a two-level λ-expression given a λ-expression and a two-level type. However, due to the well-formedness constraints in the two-level λ-calculus (in particular on the two-level types) this may result in more parameters being delayed to run-time that one ideally would like. This motivates transforming the original λ-expression so that the binding time analysis will produce a more desirable result. The transformations include changing the functionality of fixed point operators and currying.

The second group of transformations concerns the introduction of combinators and is intended to obtain more pleasant combinator expressions than those directly resulting from [14] or [16]. The transformations in this group may be characterised as *algebraic transformations* of the kind also developed for the functional language FP. These transformations are sufficiently general that they are valid under all interpretations (or at least all interpretations of interest). They are the natural counterpart at the run-time level of the transformations performed by partial evaluation at the compile-time level.

The third group of transformations consists of those transformations at the run-time level that are not valid in general. Examples include replacing an expression by an expression that produces a constant result and removing parts of expressions. The applicability of these transformations at a given program point not only depends on the semantic interpretation but also on the context in which the program point occurs. Information about the context is calculated by means of an abstract interpretation and to exploit this information for program transformation one needs a *sticky* version of the analysis (as was already discussed in Subsection 3.1).

3.8 Towards Extending TML

As should be clear from the development above the two-level metalanguage is obtained from a typed λ-calculus by imposing a distinction between the binding times *compile-time* and *run-time*. The typed λ-calculus (with product types, sum types and recursive types) is not, however, the most encompassing programming language that exists. To give but a few examples it lacks general forms of polymorphism and universal types, abstract data types and existential types, dependent products and sums, classes and inheritance and facilities for concurrent computation. This means that there are many shortcomings in the typed λ-calculus if it is to be used as a general metalanguage for semantics and evidently this shortcoming carries over to the two-level metalanguage.

To extend the two-level metalanguage so as to be a more applicable metalanguage for today's programming languages one should begin with extending the underlying λ-calculus and then introduce the two-level distinction. (The latter can be done in a very systematic way for a wide variety of languages and more than two levels as is demonstrated by the treatment of 'the **B**-level language **L**' in [NiNi90, Chapter 3].) Many of the language constructs omitted in the typed λ-calculus have been studied by other authors and [17] therefore concentrates on the communication aspects. Several well-known calculi for communication aspects exist, e.g. CSP [Hoa85] and CCS [Mil80], but these do not allow (functions and) processes as 'first-class citizens' in the same way that the λ-calculus allows functions as 'first-class citizens'. Recently, increased interest has arisen in extending CSP- and CCS- like languages with such facilites. We shall just mention [Tho89] that builds on an untyped approach. However, the development of abstract interpretation and code generation above has benefitted considerably by the presence of types and so it would be more appropriate for our purposes to use a typed approach and [17] sets about to do so.

As a very first step in this direction I present a simple programming language with types that include processes and (higher-order) functions. In particular the type of a process will indicate the type of values that can be communicated over channels but the causality between input and output, and between input/output over one channel and input/output over another channel, cannot be modelled. (Otherwise one would have to give up the static nature of the type system.) As is usual for typed languages, [17] presents an inference system for when an expression has a given type but it does not go into the existence of principal types nor into the development of a type assignment algorithm.

I also present an operational semantics for the language. It ignores all type information and is much like the operational semantics for CCS extended with the reduction rules of the λ-calculus. The distinction between 'lazy' and 'eager' positions (in a functional language) is used to express the communication possibilities allowed. The relationship between the types and the semantics is then obtained by proving that the evaluation preserves types as assigned by the inference system and that the external communications are described by the types. Traditionally a result like this is formulated as the slogan that *well-typed programs don't go wrong* [Mil78]. — I believe that [17] is the first treatment where a general notion of type is amalgamated with a language that allows processes as 'first class citizens'.

4 Methodological Issues

This section takes a closer look at a few aspects of the development that do not clearly follow the 'columnwise' classification used in Section 3. Examples are the choice of semantics, the use of a metalanguage, the role of program points and the desired degree of automation.

Operational Semantics versus Denotational Semantics

So far there has been no explicit discussion of the use of denotational semantics. At the time the present research started (as explained in Subsection 3.1) there hardly were any serious contenders to denotational semantics:

- The *axiomatic approach*, e.g. Hoare Logic's for partial correctness, was not as widely used as denotational semantics for the definition of programming languages. Furthermore, the rather indirect logical nature of the axiomatic approach presents little guidance on how to perform abstract interpretation and code generation.

- The *operational approach*, e.g. the SECD-machine, seemed rather dependent on an actual (abstract) machine architecture so that it would be hard to formulate abstract interpretation and code generation at a convenient level of abstraction.

- The *algebraic approach*, e.g. of the ADJ-group, seemed to be little more than a variant of denotational semantics but with a stronger focus on the homomorphic nature of the semantic functions.

However, since then new forms of operational semantics have been formulated. In [Plo81] Plotkin defines a notion of *Structural Operational Semantics*. Here there is a transition relation between configurations that consist of a syntactic component (essentially a textual representation of the continuation in denotational semantics) and a semantic component (e.g. the store or state of a machine). To cater for the termination of programs there also is a class of final configurations which have no syntactic component and as in denotational semantics block structure is handled using environments. A variant of Structural Operational Semantics has been popularised under the name *Natural Semantics* [CDDK86]. Here the transition relation always relates a 'two-component' configuration with a final configuration.

For the kind of imperative and functional languages upon which I have concentrated it is quite likely that natural semantics would be a strong contender to denotational semantics. This seems quite obvious for code generation because there is no need for mathematical sophistication of the kind needed for denotational semantics[8]. Examples of this approach may be found in e.g. [DaJe86] (which I supervised) and [CDDK86]. Concerning abstract interpretation it is less clear how desirable natural semantics is with respect to denotational semantics as abstract interpretation involves partial orders and fixed points in a rather natural way and so there might be the need to combine two rather different approaches. — However, it seems fair to conclude that both denotational semantics and natural semantics are promising candidates for a useful semantic metalanguage.

Structural operational semantics seems less desirable than natural semantics as long as one stays within programming languages that can be handled by natural semantics. The reason is that natural semantics is syntax directed in much the same homomorphic way that denotational semantics is whereas structural operational semantics is syntax directed in a rather more complicated way. However, if concurrent language features are added natural semantics seems too weak and it would appear that denotational semantics must be extended with powerdomains and resumptions. In this case the structural operational semantics might lead to a simpler approach than denotational semantics but further research is needed.

Concerning denotational semantics I have for the most part used continuous functions and cpo's with a least element. For technical reasons I have used monotonic functions in [3], [4] and [9] (and noncontinuous and nonmonotonic functions in [1] and [2]). The use of monotonic functions precludes recursive types but makes *some* proofs easier (as one does not have to prove continuity) but other proofs might be harder (because one cannot rely on continuity). There is no technical merit (but a pedagogical merit) to be gained from using partial continuous functions and cpo's that need not have a least element as any construction in this setting can be mimicked in the more traditional setting (and amounts to treating the least element in the 'right way').

[8]This point was indeed one of the motivations behind Plotkin's development of Structural Operational Semantics.

Programming Language Based versus Metalanguage Based

Given the ultimate desire to obtain a system for compiler construction there is hardly any alternative to basing the development on a metalanguage, i.e. on some notation to be 'understood' by the system for compiler construction. This metalanguage is likely to have certain restrictions which may vary according to the task that one wants to perform. It is therefore important to relate these restrictions in the metalanguage to concrete concepts in programming languages so as to aid the intuition of the user of such a system.

Examples of restrictions in the metalanguage, and the corresponding programming language concepts, may be found throughout the work:

- the formal constraint *contravariantly pure* may be motivated as disallowing a compiler that would generate code depending on test runs of particular pieces of code,
- the subset TMLs of TML may be motivated as disallowing storable (or first-class) procedures (and functions) and so corresponds to PASCAL-like languages much as TMLm corresponds to general functional languages, and
- the formal constraint *composite* may be motivated as a preference for a direct-style semantics rather than a continuation-style semantics.

The identification of, and study of, subsets of a metalanguage is probably the only way in which one can structure the problems to be solved in semantics directed compiling. Each subset clearly indicates abilities and limitations in various techniques. Thereby they suggest where techniques should be extended and, if this fails, they suggest where one should develop heuristics for transforming a semantics in a large subset of TML into a smaller subset for which a particular implementation technique has been developed. We already saw an example of this in Subsection 3.6 above.

Program Points versus No Program Points

Apart from [1] and [2] the papers have not incorporated program points, i.e. the results of an abstract interpretation are only recorded at the end of a program and not at any intermediate program point. As is discussed in the Conclusion of [10] one could envisage an analogous development using program points. As should be clear from the development in [2] and [18] the use of program points is *mandatory* for the practical exploitation of program transformations whose applicability depend on results of analyses.

Originally program points were abandoned in order to simplify the setup of [1] and [2] so that it would be easier to perform a development for a denotational metalanguage. It was hoped that it would be rather straightforward to re-introduce program points along the lines of [1]. However, the development in [HuYo88] shows that this need not be so. The core of the problem is that for a functional language (unlike an imperative language [2]) the order of evaluation of expressions is *not* specified by the standard semantics nor by an abstract interpretation (of the kind without program points). So when some subexpression of an operator is non-terminating it becomes nontrivial to decide what information to record for another subexpression of that operator as this may depend on the order of evaluation. — Thus further work is needed and I believe that the way in which accumulation at program points is introduced will reflect a particular evaluation strategy.

The Degree of Automation

The outlook presented in Section 2 calls for a compiler construction system that can process a semantic description in an automatic way. An underlying principle of abstract interpretation is to approximate perfect information by 'erring on the safe side'; following this principle I have for the most part selected particular subtasks in compiler construction and for each one I have identified subsets of TML for which a correct and automatic treatment could be performed. This has been used as a basis for an experimental system, the *PSI-system* [Nie87b], in which one can experiment with expressions and interpretations of the metalanguage. A front end, called *DAOS* [AnCh87], adapts the system to allow denotational definitions based on the two-level metalanguage.

An advantage of this approach is that it points directly towards the end goal and the pieces of insight obtained all rest on good foundations. A disadvantage is that the overall connection between the various subtasks may not be expressed in case the connection has not yet been fully explored. One could indeed argue that there are so many problems in the area of semantics directed compiler construction that one should focus *less* on the automation aspect and *more* on developing systematic procedures and guidelines that might help a human in constructing a correct compiler for some programming language. The presentation in [18] is to a large extent based on this point of view (and it may well be contrasted with [NiNi88b]). — It would seem that large parts of the literature on semantics directed compiling follow this approach rather than adhering strictly to the aim of 'full automation'.

Semantics versus Symbol Pushing

To me the fundamental metalanguage is TML or one of its subsets. It is thefore for TML that I formulate the notion of interpretation and parameterised semantics. However, it may not be entirely pleasant to write semantic equations in TML and so one may want convenient shorthands that can improve on this. This point of view is quite analogous to the point of view taken in the programming language (Standard) ML. Here the interest is on typed programs but mostly one can dispense with the types because there is an algorithm for *type analysis* that can insert types into (many[9]) untyped expressions.

So the impact of the work overviewed in Subsection 3.3 is to provide a shorthand for TML where semantics may be expressed as untyped λ-expressions but with their overall two-level types given. As a first step one can then use *type analysis* to transform (many) untyped λ-expressions into typed λ-expressions. As a second step one can use *binding time analysis* to transform (all) typed λ-expressions into two-level λ-expressions. And as the final step one can use *two-level λ-lifting* to transform (potentially all) λ-expressions into expressions in the two-level metalanguage.

The above point of view does not ascribe any semantic content to type analysis, binding time analysis and two-level λ-lifting. One can do so for type analysis by providing a denotational semantics for the untyped λ-calculus and then show that 'well-typed programs do not go wrong'. For binding time analysis one could aim at showing that the semantics of the two-level λ-expression is 'faithful' to itself (see [9] above). (A somewhat related approach is taken in [Mog89].) For two-level λ-lifting one can show that the original

[9]In principle the presence of recursive types means that any untyped λ-expression may be typed; this may be done using Scott's work on interpreting untyped λ-expressions as elements of a universal domain and this domain can be constructed as a recursive type. However, a concrete algorithm for type analysis [Mil78] need not succeed on all expressions.

two-level expression has the same semantics as the transformed metalanguage expression. However, even if one does show results like these this does not yield any results that hold in general for the parameterised semantics, but only results for the interpretation that corresponds to the denotational semantics of the untyped λ-calculus. For this reason I have chosen not to ascribe semantic content to type analysis, binding time analysis and two-level λ-lifting.

5 Related Work

The papers provide detailed comparisons with related work and some references and comparisons have already been given above. This section is therefore centered around a few central themes.

Compiler Construction Systems

Probably the first system capable of automatically processing a semantic description is P. Mosses' SIS system [Mos79][10]. This system accepts a form of denotational semantics but is not really a compiler construction system because it does not have any (direct) ability to generate code. Also there is no general treatment of correct program analyses or correctness of code generation. However, one can develop a semantics that depends on the meaning of certain primitives and in this way one obtains a notion of parameterised semantics. Much of the pragmatic usefulness of the system comes from the use of partial evaluation to specialise the semantics once meanings of some of the primitives are provided. The distinctions that I obtain in an explicit way (through the use of underlining etc. to distinguish various binding times) thus arise here in an implicit way.

Much work on compiler constrution systems has centered around the use of attribute grammars and in this work the semantic issues seldom play a major role. As a tool in specifying compilers the use of attribute grammars has been rather succesful in terms of the quality of the compilers produced. An example system is L. Paulson's system [Pau84] that uses a metalanguage (called *semantic grammars*) that is more powerful than those found in most other systems. As is explained in [11] the two-level metalanguage may be regarded as an extension of attribute grammars in general and semantic grammars in particular. Another example is [Wil81] that pays considerable attention to the use of program analyses for program transformations.

Another major line of research has used combinators of the kind usually used in forms of algebraic semantics. An early paper in this area is by the ADJ-group [TWW81] and relates to a previous paper by F.L. Morris [Mor73]. Much later work have built on a selection of combinators developed by P. Mosses under the name *action semantics*, e.g. [Mos80]. This also includes work by N.D. Jones and H. Christiansen [ChJo82] and work by P. Lee and U. Pleban [LePl86]. In some of this work, notably [TWW81] and [Mos80], correctness is considered but only for a rather small class of programming languages. In order to be useful in a system for compiler construction the choice of combinators used in the system should be rather fixed and should ideally not depend on the programming language to be implemented. This aim applies to P. Mosses' work on *action semantics* but, unfortunately, is not always maintained by those who use his ideas. An example is [LePl86] that would allow defining combinators corresponding to each construct in

[10]The development in [Nie79] was performed using SIS.

the programming language to be implemented. This means that too much work is being placed on the user of the system and, more importantly, the user has to conduct all proofs on his own as there is too little structure on which to develop a theory of correctness. (Accordingly, [LePl86] does not go deeply into the issue of correctness.)

Another approach to the development of a system for compiler construction builds on the use of partial evaluation and mixed computation, e.g. [JSS85]. The idea is here that a semantics for a language should be given in terms of an interpreter. Once the interpreter is supplied with a program it may be specialised using partial evaluation much as in the SIS-system overviewed above. However, this method can only be considered to generate 'code' for the machine upon which the interpreter is implemented and so does not lead to a compiler construction system in the sense of Section 2.

Many other researchers have studied the development of compiler construction systems, e.g. [JoSc80], [Wan82], [Ras82] and [Set83], but we shall refer to the papers (in particular [11]) for comparisons.

Correctness of Code Generation

One of the first, and most impressive, developments of a correct implementation by means of code generation was given by R. Milne and C. Strachey [MiSt76] but unfortunately most researchers have found that their insights were not presented in a sufficiently abstract way so as to be applicable in other settings. This work may be contrasted with the algebraic approaches mentioned above, e.g. [TWW81] and [Mos80], where there is a nice structure on the proofs but where only rather simple programming languages are handled. As stated in [11] the aim of my work on code generation was to bridge this gap. One difference between my work and that of [MiSt76] (and [Wan82]) is that I generate code from a direct-style semantics whereas they generate code from a continuation-style semantics.

Another very promising approach to the construction of correct code generation is presented in [CDDK86] and [Des86]. Here natural semantics is used as the semantic platform and an example is given of compiling a subset of the programming language ML into the Categorical Abstract Machine. As was said above one advantage of this approach with respect to a denotational approach (like mine) is that a machine is almost inevitably given in an operational way and one then does not have to face the problems involved in relating denotational and operational semantics. It would therefore be interesting to investigate whether the notion of two levels of binding times could be used to give a 'parameterised' form of a transition relation. (In particular because structural operational semantics and natural semantics are becoming increasingly more used for semantics.)

It should not be forgotten, however, that much work in the λ-calculus has centered around the use of denotational semantics because proofs here turn out to be more 'abstract' than those using an operational semantics (in the form of β-reduction). This is not really in conflict with what was said in the previous paragraph because the operational semantics of the λ-calculus is rather close to structural operational semantics and we already noted the different syntax directed behaviour of 'natural semantics' and 'structural operational semantics'.

Abstract Interpretation

The pioneering work in this area was done by P. Cousot and R. Cousot by extending the motone data flow analysis frameworks of J.B. Kam and J.D. Ullman. This centered around flow charts and the use of *collecting semantics* (static semantics in their terminology and *sticky lifted standard semantics* in present terminology) and *pairs of adjoined functions* to represent the meaning of the properties used in the analyses. A denotational formulation using these ideas was first attempted by V. Donzeau-Gouge [Don78] and motivated [1] (and later e.g. [5]) where a more comprehensive treatment of abstract interpretation was obtained. One should also note the high-level formulations of data flow analysis developed by B.K. Rosen, e.g. [Ros77] and [Ros80], but as in the work of J.B. Kam and J.D. Ullman the semantic considerations hardly enter the picture.

As is discussed at great length in Section 2 of [15] other authors have had other approaches. In [BHA86] a motivation was to avoid relying on the existence of the collecting semantics by using a standard semantics instead and thus replace abstraction functions with what I call representation functions[11]. (The terminology used in [BHA86] is different.) In [MyJo86] a motivation was to dispense with abstraction and concretization functions altogether and so use relations instead and [Myc87] went further along this line. This prompted [9] that generalised the predicate 'contravariantly pure' to 'level-preserving' and conducted a development along the lines of [5] for the analyses considered in [BHA86]. In [15] the various views are brought even closer to one another but in response to the slogan 'abstract interpretation = correctness' of [Myc87] the paper proposes the slogan 'abstract interpretation = correctness + induced analyses + implementable analyses'.

Other approaches to abstract interpretation may be found in e.g. [AbH87] and [Hug88] and the special case of strictness analysis is treated in e.g. [Hug86], [Mau86] and [Dyb87]. — The main hallmark of my approach is that it is based on a metalanguage (so as to obtain generality) and has a strong and formal semantic content.

Binding Times

The distinction between compile-time and run-time is fundamental in most work on semantics directed compiling, e.g. [App85], [Cli84], [Pau84] and [Ras82]. As an example, M.R. Raskovsky [Ras82] presents a system that rewrites a traditional denotational semantics into a compiler. This is based on a distinction between compile-time and run-time domains but the distinction is not expressed as clearly as in the two-level metalanguage and relies on a rather fixed set of semantic domains. The correctness of the transformations is not proved nor is there a precise statement of the class of semantic descriptions for which they are applicable.

Also the work on 'staging transformations' by U. Jørring and W. Scherlis [JøSc86] builds on an informal distinction between 'stages' that roughly corresponds to my notion of binding times. The staging transformations then aim at seperating the stages from one another much as if one would collect all compile-time computations at the outermost level so that none were embedded inside run-time computations. A similar aim may be found in [MoWa88].

Binding time analysis has also been formulated by the DIKU-group in connectection with their work on partial evaluation and mixed computation [JSS85]. The result of the binding time analysis is recorded as annotations on the program much as I have used underlinings etc. A major difference is that whereas my work has been based on a typed λ-calculus ([12] and [13]) the work of the DIKU-group originally considered only untyped first-order programs. Also [JSS85] performed binding time analysis by means of

[11]This is *not* the same concept as the representation functions discussed in Subsection 3.6.

abstract interpretation whereas [12] and [13] performed it by means of a 'static' analysis on expressions. Later work has decreased the gap between the approaches [Mog89].

6 Future Work

A major new aim in my future work will be to extend the applicability of the TML-based development so that also parallel language constructs may be handled. This will take place within an ESPRIT BRA project on *Provably Correct Systems* in the context of developing a compiler for a subset of OCCAM-2 into a transputer-like architecture. To express the semantics of OCCAM-2 and the transputer it is anticipated that structural operational semantics will be used. The correctness proof for code generation in [11] will therefore have to be modified so as to take this difference into account. It therefore remains to be seen what forms of TML will prove most useful, whether also the compile-time level in TML should have an operational semantics (rather than a denotational semantics) and whether one can develop a notion of 'parameterised structural operational semantics' that is as versatile as the similar notion for denotational semantics.

Acknowledgments

I have learned much from my supervisors: Peter Mosses ([Nie79]), Neil Jones (for M.Sc. [Nie81]) and Gordon Plotkin (for Ph.D. [Nie84]). In recent years a grant from The Danish Natural Science Research Council and a fellowship from The Technical University of Denmark (tenable in Dines Bjørner's group) have been very encouraging. During the years my wife (Hanne Riis Nielson) has provided valuable personal and professional support.

References

[AbH87] S.Abramsky & C.Hankin: *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[AnCh87] K.B.Andersen & S.R.Christensen: DAOS - Konstruktion af et Dynamisk Automatisk Oversætter System, (in Danish), *report IR 87-05, Aalborg University Centre*, 1987.

[App85] A.W.Appel: Semantics-directed code generation, *Proc. 12th ACM Conf. on Principles of Programming Languages* 315–324, 1985.

[ASU86] A.V.Aho & R.Sethi & J.D.Ullman: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[BHA86] G.L.Burn & C.Hankin & S.Abramsky: Strictness analysis for higher-order functions, *Science of Computer Programming* 7 249–278, 1986.

[Car83] L.Cardelli: The functional abstract machine, *Bell Labs. Technical Report TR-107*, 1983.

[CCM87] G.Cousineau & P.L.Curien & M.Mauny: The Categorical Abstract Machine, *Science of Computer Programming* 8 173–202, 1987.

[CDDK86] D.Clément & J.Despeyroux & T.Despeyroux & G.Kahn: A Simple Applicative Language: Mini-ML, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 1986.

[ChJo82] H.Christiansen & N.D.Jones: Control flow treatment in a simple semantics-directed compiler generator, *Formal Description of Programming Concepts II*, D.Bjørner (ed.), North-Holland, 1982.

[Cli84] W.Clinger: The SCHEME 311 Compiler. An exercise in denotational semantics, *Proc. ACM Conference on LISP and Functional Programming* 356–364, 1984.

[CoCo79] P.Cousot & R.Cousot: Systematic design of program analysis frameworks, *Proc. 6th ACM Symp. Principles Prog. Lang.* 269–282, 1979.

[DaJe86] M.Dam & F.Jensen: Compiler Generation from Relational Semantics, *Proceedings ESOP 1986*, Springer Lecture Notes in Computer Science **213** 1–29, 1986.

[CuFe58] H.B.Curry & R.Feys: *Combinatory Logic* vol. **1**, North-Holland, 1958.

[Des86] J.Despeyroux: Proof of Translation in Natural Semantics, *Symposium on Logic in Computer Science*, 1986.

[Don78] V. Donzeau-Gouge: Utilisation de la Sémantique dénotationelle pour l'étude d'interpretations non-standard, *IRIA Report 273*, 1978.

[Dyb85] P.Dybjer: Using domain algebras to prove the correctness of a compiler, *Proc. STACS 1985*, Springer Lecture Notes in Computer Science **182** 98–108, 1985.

[Dyb87] P.Dybjer: Inverse Image Analysis, *Proc. ICALP 87*, Springer Lecture Notes in Computer Science **267** 21–30, 1987.

[Ger75] S.L.Gerhart: Correctness-preserving program transformations, *Proceedings 2nd ACM Symposium on Principles of Programming Languages* 54–66, 1975.

[Gor79] M.J.C.Gordon: *The Denotational Description of Programming Languages, an Introduction*, (Springer, Berlin), 1979.

[HeAs80] M.C.B.Hennessy & E.A.Ashcroft: A Mathematical Semantics for a Nondeterministic Typed λ-Calculus, *Theoretical Computer Science* **11** 227–245, 1980.

[Hoa85] C.A.R.Hoare: *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hug82] J.Hughes: Supercombinators: a new Implementation Method for Applicative Languages, *Proc. of 1982 ACM Conf. on LISP and Functional Programming*, 1982.

[Hug86] J.Hughes: Strictness detection in non-flat domains, *Proc. Programs as Data Objects*, Springer Lecture Notes in Computer Science **217**, 1986.

[Hug88] J.Hughes: Backwards Analysis of Functional Programs, *Partial Evaluation and Mixed Computation*, D.Bjørner, A.P.Ershov and N.D.Jones (eds.), 187–208, North-Holland, 1988.

[HuYo88] P.Hudak J.Young: A Collecting Interpretation of Expressions (without Powerdomains), *Proceedings of the 15th ACM Symposium on Principles of Programming Languages* 107–118, 1988.

[JoNi91] N.D.Jones & F.Nielson: Abstract Interpretation, invited paper (in preparation) for *The Handbook of Logic in Computer Science*, North-Holland, 1991.

[JoSc80] N.D.Jones & D.A.Schmidt: Compiler Generation from Denotational Semantics, *Proc. Semantics Directed Compiler Generation*, Springer Lecture Notes in Computer Science **94** 70–93, 1980.

[JSS85] N.D.Jones & P.Sestoft & H.Søndergaard: An experiment in Partial Evaluation: The Generation of a Compiler Generator, *Proceedings of Rewriting Techniques and Applications*, Springer Lecture Notes in Computer Science **202**, 1985.

[JøSc86] U.Jørring & W.L.Scherlis: Compilers and Staging Transformations, *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986.

[KaUl77] J.B.Kam & J.D.Ullman: Monotone data flow analysis frameworks, *Acta Informatica* **7** 305–317, 1977.

[LePl86] P.Lee & U.Pleban: On the use of LISP in implementing denotational semantics, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* 233-248, 1986.

[Mau86] D.Maurer: Strictness computation using generalized λ-expressions, *Proc. Programs as Data Objects*, Springer Lecture Notes in Computer Science **217**, 1986.

[Mil75] R.Milner: Processes: A Mathematical Model of Computing Agents, *Logic Colloquium 1973* 157–174, North-Holland, 1975.

[Mil78] R.Milner: A theory of type polymorphism in programming, *J. Comput. System Sci.*, 1978.

[Mil80] R.Milner: *A Calculus of Communicating Systems*, Springer Lecture Notes in Computer Science **92**, 1980.

[MiSt76] R.Milne & C.Strachey: *A Theory of Programming Language Semantics*, Chapman and Hall, 1976.

[Mog89] T.Mogensen: Binding time analysis for higher order polymorphically typed languages, *TAPSOFT 1989*, Springer Lecture Notes in Computer Science, 1989.

[Mor73] F.L.Morris: Advice on structuring compilers and proving them correct, *Proc. ACM Conf. on Principles of Programming Languages* 144–152, 1973.

[Mos79] P.D.Mosses: SIS — Semantics Implementation System: reference manual and user guide, *Technical Report, Aarhus University*, 1979.

[Mos80] P.D.Mosses: A constructive approach to compiler correctness, *Proceedings ICALP 1980*, Springer Lecture Notes in Computer Science **85** 449–462, 1980.

[MoWa88] M.Montenyohl & M.Wand: Correct Flow Analysis in Continuation Semantics, *Proc. ACM Conference on Principles of Programming Languages*, 1988.

[Myc81] A.Mycroft: Abstract Interpretation and Optimising Transformations for Applicative Programs, *Ph.D.-thesis*, University of Edinburgh, 1981.

[Myc87] A.Mycroft: A Study on Abstract Interpretation and 'Validating Microcode Algebraically', *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin (eds.), 199–218, Ellis Horwood, 1987.

[MyJo86] A.Mycroft & N.D.Jones: A relational framework for abstract interpretation, *Programs as Data Objects*, Springer Lecture Notes in Computer Science **217**, 1986.

[Nie79] F.Nielson: Compiler Writing Using Denotational Semantics, *DAIMI TR-10*, Aarhus University, 1979.

[Nie81] F.Nielson: Semantic Foundations of Data Flow Analysis, *M.Sc. thesis, DAIMI PB-131*, Aarhus University, 1981.

[Nie84] F.Nielson: Abstract Interpretation Using Domain Theory, *Ph.D.-thesis, CST-31-84*, University of Edinburgh, 1984.

[NiNi86a] F.Nielson & H.R.Nielson: Comments on Georgeff's 'Transformation and Reduction Strategies for Typed Lambda Expressions', *ACM Transactions on Programming Languages and Systems* 8 *3* 406–407, 1986.

[NiNi86b] F.Nielson & H.R.Nielson: Code Generation from Two-Level Denotational Meta-Languages, *Programs as Data Objects*, Springer Lecture Notes in Computer Science **217** 192–205, 1986.

[NiNi86c] H.R.Nielson & F.Nielson: Pragmatic Aspects of Two-Level Denotational Meta-Languages, *Proceedings ESOP 1986*, Springer Lecture Notes in Computer Science **213** 133–143, 1986.

[Nie86a] F.Nielson: A Bibliography on Abstract Interpretation, *ACM SIGPLAN Notices* **21** *5* 31–38, 1986 and *Bulletin of the EATCS* **28**, 1986.

[Nie86b] F.Nielson: Correctness of Code Generation from a Two-Level Meta-Language (Extended Abstract), *Proceedings ESOP 1986*, Springer Lecture Notes in Computer Science **213** 30–40, 1986.

[Nie87a] F.Nielson: Strictness Analysis and Denotational Abstract Interpretation (Extended Abstract), *ACM Conference on Principles of Programming Languages* 120–131, 1987.

[Nie87b] H.R.Nielson: The Core of the PSI-system, *report IR 87-02, Aalborg University Centre*, 1987.

[NiNi88a] H.R.Nielson & F.Nielson: Automatic Binding Time Analysis for a typed λ-Calculus (Extended Abstract), *ACM Conference on Principles of Programming Languages* 98–106, 1988.

[NiNi88b] F.Nielson & H.R.Nielson: The TML-approach to compiler-compilers, *Report ID-TR 1988-47*, The Technical University of Denmark, 1988.

[NiNi89] H.R. Nielson & F.Nielson: The Mixed λ-Calculus and Combinatory Logic (an overview), *Proceedings of the International Conference on Computing and Information*, North-Holland, 39–45, 1989.

[NiNi90] F.Nielson & H.R.Nielson: *Two-Level Functional Languages*, lecture notes.

[Pau84] Compiler generation from denotational semantics, *Methods and Tools for Compiler Construction*, B.Lorho (ed.), 219–250, Cambridge University Press, 1984.

[Plo77] G.D.Plotkin: LCF considered as a programming language, *Theoretical Computer Science* **5** *3*, 1977.

[Plo76] G.D.Plotkin: A Powerdomain Construction, *Siam J. Comput.* **5** *3* 452–487, 1976.

[Plo80] G.D.Plotkin: Lambda-definability in the full type hieararchy, *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, (J.P.Seldin and J.R.Hindley, eds.), Academic Press, 1980.

[Plo81] G.D.Plotkin: A Structural Approach to Operational Semantics, *Report DAIMI FN-19*, Aarhus University, 1981.

[Plo82] G.D.Plotkin: A Powerdomain for Countable Nondeterminism, *Proceedings ICALP 1982*, Springer Lecture Notes in Computer Science **140** 418–428, 1982.

[Ras82] M.R.Raskovsky: Denotational semantics as a specification of code generators, *Proc. SIGPLAN 1982 Symp. on Compiler Construction* 230–244, 1982.

[Rey74] J.C.Reynolds: On the Relation between Direct and Continuation Semantics, *Procedings 2nd ICALP 1974*, Springer Lecture Notes in Computer Science **14** 141-156, 1974.

[Ros77] B.K.Rosen: High-level data flow analysis, *Communications of the ACM* **20** *10* 712–724, 1977.

[Ros80] B.K.Rosen: Monoids for rapid data flow analysis, *SIAM Journal of Computing* **9** 159–196, 1980.

[Sch24] M.Schoenfinkel: Über die Bausteine der mathematischen Logik, *Mathematische Annalen* **92**, 1924.

[Set83] R.Sethi: Control flow aspects of semantics directed compiling, *ACM TOPLAS* **5** *4* 554–595, 1983.

[SmPl82] M.B.Smyth & G.D.Plotkin: The category-theoretic solution of recursive domain equations, *SIAM J. Comput.* **11** *4*, 1982.

[Smy78] M.B.Smyth: Powerdomains, *J. Comput. System Sci.* **16** 23–36, 1978.

[Ten81] R.D.Tennent: *Principles of Programming Languages*, Prentice-Hall, 1981.

[Tho89] B.Thomsen: A Calculus of Higher Order Communicating Systems, *Proceedings of the 1989 ACM Conference on Principles of Programming Languages* 143–154, 1989.

[Tur79] D.A.Turner: Another algorithm for bracket abstraction, *The Journal of Symbolic Logic* 44, 1979.

[TWW81] J.W. Thatcher & E.G. Wagner & J.B. Wright: More on advice on structuring compilers and proving them correct, *Theoretical Computer Science* **15** 223–249, 1981.

[Wan82] M.Wand: Deriving target code as a representation of continuation semantics, *ACM TOPLAS* 4 *3* 496–517, 1982.

[Wil81] R.Wilhelm: Global Flow Analysis and Optimization in the MUG2 Compiler Generating System, *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D.Jones (eds.), 132–159, Prentice-Hall, 1981.

A Errata

Errata in [5]:

- Page 17: Line 12 should read:

  be proved by structural induction on contravariantly pure types ct)

Errata in [8]:

- Page 251, left column: Insert the following line between lines 8 and 9:

  $| \ \mathrm{case}(e_1,\ldots,e_k) \ | \ \underline{\mathrm{in}}_j \qquad\qquad (\text{for } \underline{+})$

- Page 254, right column: Lines 28 and 29 should read:

$$\underline{\mathrm{K}}_e(\mathrm{fix}_{ct+ct'}) = \lambda G.\ \mathrm{is}_1(G\bot) \rightarrow \mathrm{in}_1(\underline{\mathrm{K}}_e(\mathrm{fix}_{ct})(\mathrm{out}_1 \mathrm{o} G \mathrm{o} \mathrm{in}_1)),$$
$$\mathrm{in}_2(\underline{\mathrm{K}}_e(\mathrm{fix}_{ct'})(\mathrm{out}_2 \mathrm{o} G \mathrm{o} \mathrm{in}_2))$$

Errata in [10]:

- Page 219: Line 7: '[Con77a]' $\mapsto$ '[Cou77a]'.
- Page 219: Line 10: '[HBP86]' $\mapsto$ '[Han86]'.
- Page 228: Figure 1: Insert an arrow from 'Programming language' to 'metalanguage'.
- Page 238: Line 8 from the bottom: '$\downarrow$2' $\mapsto$ '$\downarrow$1'.
- Page 238: Line 3 from the bottom: '$\sqsubseteq$' $\mapsto$ '$\leq$'.

Errata in [11]:

- Page 80: Last line: '(d)' $\mapsto$ '$(\bot)(r^u_{n_d}(d))$'.

Errata in [14]:

- Page 338: Line 11 should read:

$$\varphi(\mathrm{tt},\mathrm{penv}) = \begin{cases} \mathrm{penv} & \text{if penv} \neq () \\ ((x_a,\mathrm{tt}_1)) & \text{if penv} = () \text{ and tt} = \mathrm{tt}_1 \underline{\rightarrow} \mathrm{tt}_2 \end{cases}$$

Errata in [15]:

- Page 200: Line 8: '$\times \cdots \times$' $\mapsto$ '$\otimes \cdots \otimes$'.

B Terminology

There is some confusion in the literature about the meaning of the term 'collecting semantics'. To clarify this let us first recall the terminology used in [MiSt76]: The *standard semantics* of a programming language is a semantics that incorporates no implementation oriented details; in the *store semantics* the environment is then made part of the state and in the *stack semantics* the environment behaves much like a symbol table in a compiler. Let us now extend this terminology by saying that a semantics is *lifted* it if operates on sets of values rather than actual values and that a semantics is *sticky* if the values or states that reach given program points are accumulated at those points. For the flow chart model the program points would naturally correspond to the arcs so that there is no need to distinguish sharply between sticky and non-sticky versions but this is necessary for a denotational formulation.

The development of abstract interpretation in [CoCo79] starts with a standard (or store) semantics of a flow chart. It next considers a so-called *static semantics* which merely is a (sticky) lifted standard (or store) semantics. In the denotational formulation given in [1] and [2] a similar route is taken but as an intermediate step a so-called *collecting semantics* is needed; this is merely a sticky standard (or store) semantics, i.e. a non-lifted version of the static semantics.

Unfortunately, this use of the term 'static semantics' conflicts with its use in the distinction between static and dynamic semantics. Some papers therefore use the term 'collecting semantics' to mean 'static semantics', i.e. (sticky) lifted standard (or store) semantics, and this includes [3], [4], [5], [6], [7], [9], [10] and [15].

Teoretiske aspekter af Semantik-baseret Sprog-implementation

Flemming Nielson
Datalogisk Afdeling
Aarhus Universitet
Ny Munkegade
8000 Aarhus C

Der gives et sammendrag af teoretiske aspekter involveret i implementationen af programmeringssprog direkte fra en semantisk specifikation. Dette involverer elementerne *abstrakt fortolkning* (der er en ramme for programanalyser), *kodegenerering* og *program-transformationer* og det væsentlige sigte har været at sikre *korrektheden* af disse.

1 Introduktion

Dette er en sammenfatning af artikler der er indleveret med henblik på erhvervelse af graden *doctor scientarum (dr. scient.)*. En engelsk version findes andetsteds.

Afsnit 2 giver en kort oversigt over det område af datalogien som mit arbejde omhandler. Motiveret heri giver afsnit 3 en mere detaljeret gennemgang af de enkelte artikler. Der gøres herunder rede for motivationen bag de enkelte artikler, de resultater der er opnået, de væsentlige sammenhænge og forskelle mellem de forskellige artikler og der gives sammenligninger med litteraturen[1]. Det vil fremgå at en bred formulering af problemstillingen løbende er blevet indsnævret med henblik på at kunne opnå resultater for en klasse af programmeringssprog og ikke blot for enkelte legetøjssprog. Afsnit 4 indeholder så en diskussion af nogle af de grundlæggende valg vedrørende forskningens metoder og retning. Endelig giver afsnit 5 yderligere sammenligninger med litteraturen og afsnit 6 beskriver kort et aspekt af min kommende forskning.

2 Emneområdet

Emneområdet for forskningen er udviklingen af systemer der automatisk kan konstruere (dele af) oversættere. Et af de væsentligste gennembrud inden for dette område har været udviklingen af såkaldte parser-generatorer der konstruerer parsere ud fra en grammatik for sproget: Moderne LALR(1) baserede systemer giver nu så effektive parsere at de

[1]Det bør her erindres at denne sammenfatning er skrevet i juni 1989.

kan benyttes i praktisk anvendelige oversættere. På lignende måde kan man stræbe efter et system der kan konstruere oversættere ud fra en semantik for sproget. Disse oversættere skal være tilstrækkeligt 'gode' til at kunne bruges i praksis. Dette problem er imidlertid betydeligt sværere end konstruktionen af parsere og er endnu ikke blevet løst på en tilfredsstillende måde.

Kvaliteten af et system der udvikler oversættere involverer bla.:

- *korrekthed:* at oversætterne genererer kode der opfører sig som ventet, dvs. har samme mening som det program der oversættes,

- *effektivitet:* at kodens effektivitet er sammenlignelig med effektiviteten af kode der produceres på anden vis,

- *anvendelighed:* at semantiske specifikationer af programmeringssprog skrives i en naturlig notation og at mange semantiske specifikationer tillades.

Korrekthed kræver endvidere at oversætteren altid terminerer og at selve oversætter-generatoren terminer. Effektivitet omfatter også at oversætteren er rimeligt hurtig og at oversætter-generatoren ikke er uacceptabelt langsom; dette vil imidlertid ikke indgå i det følgende da det er sekundære problemstillinger i forhold til det at sikre at selve koden er rimeligt effektiv. Endelig kan anvendeligheden af en notation afhænge af den sprogklasse man ser på, således at een notation kan være hensigtsmæssig for imperative sprog mens en anden notation kan være hensigtsmæssig for funktionalsprog etc.

For at kunne opnå effektiv kode kan man benytte sig af program-analyser og program-transformationer som i sædvanlig oversætterkonstruktion. Korrekthed omfatter så:

- at sikre at en simpel *kodegenererings-strategi* er korrekt,

- at sikre at *program-analysernes* resultater er korrekte, og

- at sikre at *program-transformationerne* bevarer programmers semantik.

Analyserne og transformationerne kan anvendes både på kildeteksten og objektkoden. Som i sædvanlig oversætterkonstruktion betyder det at selve kodegenereringen ofte er relativt simpel idet en række mangler ved koden kan repareres ved efterfølgende program-transformationer ligesom programmet kan have været transformeret med henblik på at lette kodegenereringen. De program-transformationer der er af interesse i oversætter-konstruktion bevarer imidlertid ikke altid programmers mening. Der er derfor behov for program-analyser som sikrer at transformationerne kun foretages i sammenhænge hvor hele programmets mening bevares.

3 Oversigt

Med udgangspunkt i ovennævnte beskrivelse af emneområdet giver dette afsnit en oversigt over de indleverede papirer. Disse er:

1. F.Nielson: A Denotational Framework for Data Flow Analysis, *Acta Informatica* **18** 265–287 (23 sider), 1982.

2. F.Nielson: Program Transformations in a Denotational Setting, *ACM Transactions on Programming Languages and Systems 7 3* 359–379 (21 sider), 1985.

3. A.Mycroft & F.Nielson: Strong Abstract Interpretation using Power Domains, *Proceedings ICALP 1983*, Springer Lecture Notes in Computer Science **154** 536–547 (12 sider), 1983.

4. F.Nielson: Towards Viewing Nondeterminism as Abstract Interpretation, *Foundations of Software Technology & Theoretical Computer Science* **3** (20 sider), 1983.

5. F.Nielson: Abstract Interpretation of Denotational Definitions, *Proceedings STACS 1986*, Springer Lecture Notes in Computer Science **210** 1–20 (20 sider), 1986.

6. F.Nielson: Tensor Products Generalize the Relational Data Flow Analysis Method, *Proceedings 4'th Hungarian Computer Science Conference* 211–225 (15 sider), 1985.

7. F.Nielson: Expected Forms of Data Flow Analysis, *Programs as Data Objects*, Springer Lecture Notes in Computer Science **217** 172–191 (20 sider), 1986.

8. H.R.Nielson & F.Nielson: Semantics Directed Compiling for Functional Languages, *ACM Conference on LISP and Functional Programming* 249–257 (9 sider), 1986.

9. F.Nielson: Strictness Analysis and Denotational Abstract Interpretation, *Information and Computation* **76** *1* 29–92 (64 sider), 1988.

   En kort version blev publiceret som [Nie87a].

10. F.Nielson: Towards a Denotational Theory of Abstract Interpretation, *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin (eds.), 219–245 (27 sider), Ellis Horwood, 1987.

11. F.Nielson & H.R.Nielson: Two-Level Semantics and Code Generation, *Theoretical Computer Science* **56** 59–133 (75 sider), 1988.

    Foreløbige versioner blev publiceret som [NiNi86b], [NiNi86c] og [Nie86b].

12. F.Nielson: A Formal Type System for Comparing Partial Evaluators, *Partial Evaluation and Mixed Computation*, D.Bjørner, A.P.Ershov and N.D.Jones (eds.) 349–384 (36 sider), North-Holland, 1988.

13. H.R.Nielson & F.Nielson: Automatic Binding Time Analysis for a typed λ-Calculus, *Science of Computer Programming* **10** 139–176 (38 sider), 1988.

    En kort version blev publiceret som [NiNi88a].

14. F.Nielson & H.R.Nielson: 2-level λ-lifting, *Proceedings CAAP & ESOP 1988*, Springer Lecture Notes in Computer Science **300** 328–343 (16 sider), 1988.

15. F. Nielson: Two-Level Semantics and Abstract Interpretation, *Theoretical Computer Science — Fundamental Studies* **69** 117–242 (126 sider), 1989.

16. H.R.Nielson & F.Nielson: Functional completeness of the mixed λ-calculus and combinatory logic, *Theoretical Computer Science* **70** 99–126 (28 sider), 1990.

17. F.Nielson: The Typed λ-calculus with First-Class Processes, *Proceedings of PARLE 1989*, Springer Lecture Notes in Computer Science **366** 357–373 (17 sider), 1989.

18. H.R.Nielson & F.Nielson: Transformations on higher-order functions, *Functional Programming and Computer Architecture*, ACM Press, 129–143 (15 sider), 1989.

- Et engelsk resume ('Theoretical Aspects of Semantics-Based Language Implementation').
- Et dansk resume (denne artikel).

Afsnit 3.1 og 3.2 giver en oversigt over de artikler der behandler 'legetøjssprog', afsnit 3.3 til 3.7 giver en oversigt over de artikler der udnytter den 'to-niveau skelnen' der beskrives i afsnit 3.3, og afsnit 3.8 omhandler udvidelse af det underliggende sprog. I nedenstående tabel vises for hvert afsnit de papirer der omtales.

3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
1	3	13	5	9	11	12	17
2	4	16	6	15	8	18	
		14	7				
			10				

3.1 Dataflow-analyse og Program-transformationer

Artiklerne i første søjle er direkte motiveret af den generelle beskrivelse af emneområdet men er restringerede til et simpelt imperativt `while`-sprog. De er udvidelser og forbedringer af [Nie81] og er yderligere motiveret i en tidligere rapport [Nie79]. I denne udviklede jeg en sekvens af denotationssemantikker som bragte en standard semantik på en form hvor den kunne bruges til at generere kode for en simpel maskine. Denne udvikling var et forsøg på at forstå Milne og Strachey's [MiSt76] imponerende udvikling af en *standard semantik*, *'store' semantik* og *'stack' semantik* for et ALGOL-68 lignende programmeringssprog med henblik på at foretage en korrekt implementation af dette sprog. Det blev herved klart at

- *optimaliserings-aspekterne*, dvs. brugen af program-analyser og program-transformationer endnu ikke var inkorporeret i [MiSt76], og
- udviklingen i [MiSt76] endnu ikke var i en form hvor man *automatisk* kunne transformere denotationssemantikker således som det ville være påkrævet i et system til oversætterkonstruktion.

Den første observation motiverede [1] hvor jeg formulerer *abstrakt fortolkning* [CoCo79] ved hjælp af en 'continuation style' parametriseret denotationssemantik for et `while`-sprog. Udgangspunktet var en *'store' semantik* fordi de frie variable (eller højere ordens funktioner) i en *standard semantik* ikke modsvarede nogen konstruktion i de flowcharts for hvilke abstrakt fortolkning og dataflow-analyse tidligere var blevet udviklet. Dette er imidlertid ikke udtryk for nogen væsentlig indskrænkning da [MiSt76] viser hvorledes en standard semantik kan omskrives til en 'store' semantik. En mangel ved en 'store' semantik er imidlertid at den ikke omhandler nogen programpunkter. Disse er en væsentlig ingrediens i formuleringen af abstrakt fortolkning for flowcharts så dette leder til at definere en såkaldt *collecting semantik* (eller *'sticky store' semantik* [10]) der omhandler programpunkter og hvor mængden af værdier der når et givet programpunkt opsamles for hvert programpunkt. Man kan imidlertid kun udtrykke en svag form for korrekthed mellem den oprindelige 'store' semantik og den definerede collecting semantik så en svaghed ved denne tilgangsvinkel er at man til en vis grad skal 'tro på' at de to semantikker hænger naturligt sammen. (Det kommer vi imidlertid tilbage til nedenfor.)

Næste trin er så at formulere en såkaldt statisk semantik (eller *'sticky lifted store' semantik* [10] — se Appendix B) der i det væsentlige opfører sig som collecting semantikken men på mængder af argumentværdier i stedet for enkelte argumentværdier. Denne semantik kan præcist relateres til collecting semantikken og kan således opfattes som en bro til en klasse af *inducerede semantikker* hvor mængder af værdier erstattes af elementer i fuldstændige gitre ('complete lattices') der beskriver mere approximative informationer. En induceret semantik kan relateres til den statiske semantik ved hjælp af par af *adjungerede funktioner* der præcist angiver hvilke mængder af værdier som en approximativ information beskriver.

Det væsentligste formål med [1] er sammenhængen mellem den denotationssemantiske formulering af abstrakt fortolkning og traditionel dataflow-analyse. For at formulere dette er der behov for endnu en semantik der konstruerer et sædvanligt flowchart svarende til et `while`-program. Jeg viser så at den givne formulering af abstrakt fortolkning beskriver samme løsninger som *MOP*-løsningerne i dataflow-analyse. Endvidere viser jeg at (de mindre præcise men lettere beregnelige) *MFP*-løsninger kan opnås ved at modificere den inducerede semantik (bla. ved at skifte fra 'continuation style' til en form for 'direct style').

Som nævnt ovenfor er det svage led i denne udvikling at der ikke kan gives en præcis karakterisation af collecting semantikken i forhold til den 'store' semantik der var udgangspunktet. Den eneste måde dette kan forbedres på er formentligt ved at vise at resultaterne fra collecting semantikken (eller ækvivalent hermed, en induceret semantik) kan bruges til at validere program-transformationer således at det oprindelige og det transformerede program har samme mening *i den oprindelige 'store' semantik*. Det er her værd at bemærke at der ikke ville være nogen problemer hvis program-transformationerne altid bevarede programmers semantik idet der da heller ikke ville være behov for program-analyser.

I [2] betragtes to klasser af program-transformationer. Een klasse afhænger af såkaldte *forlæns* ('forward' [ASU86]) og *første-ordens*[2] program-analyser. Dermed menes at programmet analyseres i *samme retning* som 'flow of control' og at de approximative informationer direkte beskriver data (dvs. mængder af værdier). For denne slags program-transformationer viser jeg at man kan bruge collecting semantikken til at validere program-transformationer der ikke nødvendigvis bevarer programmers mening. Af tekniske årsager formuleres dette ud fra en version af collecting semantikken hvor 'continuations' er blevet fjernet som beskrevet i [MiSt76].

Den anden slags program-transformationer afhænger af *baglæns* og *anden-ordens* program-analyser. Dermed menes at programmet analyseres i *modsatte retning* at 'flow of control' og at de approximative informationer *ikke direkte beskriver data men snarere brugen af data.* I analogi med collecting semantikken defineres en såkaldt *'future' semantik* der opsamler 'continuations' ved hvert programpunkt. Jeg viser så at man kan sikre *partiel korrekthed* af program-transformationer ved hjælp af 'future semantikken'. Man kan opnå total korrekthed ved en dobbelt transformation. Dette motiverer et nærmere studium af en konkret analyse, *liveness analyse*, og her viser jeg at total korrekthed kan opnås på en mere direkte måde. — Disse resultater er af interesse fordi kun få papirer i litteraturen beskæftiger sig med korrekthed af program-transformationer der afhænger af

[2]Terminologien *første-ordens* versus *anden-ordens* blev indført i [2].

en forudgående program-analyse. En bemærkelsesværdig undtagelse er [Ger75] men her opnås kun partiel korrekthed (selv for forlæns og første-ordens analyser) og der er ikke automatiske teknikker som indsætter de nødvendige invarianter i `while`-løkker.

3.2 Stærk Abstrakt Fortolkning

Collecting semantikken ovenfor brugte mængder af værdier. Brugen af potensmængder ('powersets'), med tilknyttet partiel ordning $\subseteq$, er imidlertid ikke uproblematisk i forbindelse med anvendelsen af domæner, med tilknyttet partiel ordning $\sqsubseteq$. I [1] og [2] gav dette anledning til at tillade ikke-kontinuerte 'continuations' og funktioner og for at sikre eksistensen af de nødvendige mindste fix-punkter var det nødvendigt eksplicit at argumentere for at visse funktionaler faktisk var kontinuerte (mht. $\sqsubseteq$).

Den 'pæne' løsning på disse problemer er normalt at anvende potensdomæner ('powerdomains') fremfor potensmængder. Tre velkendte potensdomæner er det *konvekse* potensdomæne (også kaldet 'the Plotkin powerdomain' efter sin ophavsmand [Plo76]), det *øvre* potensdomæne (også kaldet 'the Smyth powerdomain' efter sin ophavsmand [Smy78]) og det *nedre* potensdomæne (også kaldet 'the Hoare powerdomain' eller 'the relational powerdomain'). Definitionen af disse potensdomæner er simplest for flade domæner, dvs. domæner med et tælleligt antal elementer, hvoraf eet er mindre end alle de andre og de øvrige elementer er usammenlignelige. Vi skal skrive $S_\perp$ for et typisk fladt domæne hvor de usammenlignelige elementer er elementerne i S og $\perp$ er det mindste element.

Elementerne i det øvre potensdomæne $\mathcal{P}_{\mathrm{S}}(S_\perp)$ er alle de endelige delmængder af S sammen med mængden $S\cup\{\perp\}$ og den partielle ordning er $\supseteq$. Da abstrakt fortolkning omhandler approximative beskrivelser af sædvanligvis uendelige mængder af værdier betyder det at det øvre potensdomæne ikke er synderligt anvendeligt for abstrakt fortolkning. Det nedre potensdomæne $\mathcal{P}_{\mathrm{H}}(S_\perp)$ består af alle delmængder af $S\cup\{\perp\}$ der indeholder $\perp$ og den partielle ordning er $\subseteq$. Dette er isomorft med potensmængden $\mathcal{P}(S)$ og viser sig at svare til udviklingen i [1] og [2]. I denne tilgangsvinkel betragtes terminering af programmer slet ikke, dvs. ikke-terminering opfattes altid som en mulighed.

For visse analyser af programmer, specielt *strictness analyser*[3] [Myc81], er det vigtigt at være i stand til at udtrykke at terminering af programmer er sikret. Det betyder at det nedre potensdomæne ikke kan bruges til alle aspekter af strictness analyse og det er derfor naturligt at se på det konvekse potensdomæne $\mathcal{P}_{\mathrm{P}}(S_\perp)$. Elementerne er de ikke-tomme delmængder af $S_\perp$ der er endelige eller som indeholder $\perp$. Den partielle ordning er den såkaldte Egli-Milner ordning, $\sqsubseteq_{\mathrm{EM}}$. Restriktionen omkring endelighed betyder imidlertid at man ikke kan beskrive mængden S der jo svarer til en beregning der altid terminerer men kan give et vilkårligt resultat. Det betyder igen at en af analyserne i [Myc81] ikke kan valideres.

Det er derfor naturligt at søge efter et potensdomæne hvor en mængde som S tillades. I [3] defineres potensdomænet $\mathcal{P}_{\mathrm{N}}(S_\perp)$ der har alle ikke-tomme delmængder af $S_\perp$ som elementer og den partielle ordning er Egli-Milner ordningen, $\sqsubseteq_{\mathrm{EM}}$. Da der nu tillades uendelige mængder uden $\perp$ betyder det at de udvidelser af funktioner som et potensdomæne giver anledning til ikke længere kan garanteres at være kontinuerte omend de altid er monotone. Samtidig eksisterer potensdomænet ikke på domæner af uendelig højde. (I [4] gives generelle betingelser for hvornår potensdomænet eksisterer.) For at sikre eksistensen af mindste fix-punkter for monotone funktioner er det derfor nødvendigt at arbejde med domæner hvor alle kæder (og ikke blot tællelige kæder) har en mindste øvre grænse. Det

[3]Den tilhørende program-transformation udskifter call-by-name med call-by-value.

følger af den manglende kontinuitet af funktions-udvidelser at potensdomænet afviger fra det generaliserede potensdomæne der beskrives i [Plo82] ligesom det selvfølgelig afviger fra de tre tidligere omtalte potensdomæner.

Skønt Egli-Milner ordningen er nødvendig for at opnå den nødvendige domæne-struktur er den ikke hensigtsmæssig til at udtrykke at een approximativ information er mere approximativ end en anden. Det leder frem til at arbejde med *udvidede domæner*, dvs. domæner med en ekstra partiel ordning. For potensdomæner er den oprindelige partielle ordning så Egli-Milner ordningen mens den ekstra partielle ordning er mængde-inklusion, $\subseteq$. Dette er meget tæt på de såkaldte nondeterministiske cpo'er der bruges i [HeAs80] til at give en denotationssemantik for nondeterministiske programmer.

Det *nye* er så at abstrakt fortolkning formuleres for par af abstraktions- og konkretisations-funktioner der er monotone i henhold til begge ordninger men kun adjungerede i henhold til den ekstra partielle ordning. (Derudover er der visse tekniske betingelser der også skal være opfyldt.) Formuleringen af abstrakt fortolkning indebærer at man kan bevise korrekthed af program-analyser samt at man kan konstruere såkaldte inducerede analyser.

Dette bruges til at validere de to strictness analyser som formuleres i [Myc81] for et simpelt sprog bestående af rekursive ligninger. Det indebærer bla. at den ekstra partielle ordning skal fortolkes på forskellig vis. Når den fortolkes som den sædvanlige Scott partielle ordning, $\sqsubseteq$, får man en teori for abstrakt fortolkning der meget svarer til den der er i [1] og [2]. — Den væsentligste indsigt som dette giver er at Scott ordningen, $\sqsubseteq$, ikke er den samme som den naturlige partielle ordning for abstrakt fortolkning, $\subseteq$. Dette blev allerede fremført i [1] men baseret på en mere intuitiv forklaring.

Sammenhængen mellem de matematiske hjælpeværktøjer der bruges til at beskrive nondeterminisme og abstrakt fortolkning leder naturligt frem til at overveje om nondeterminisme kan opfattes som en form for abstrakt fortolkning: Sædvanligvis forklares nondeterminisme ud fra behovet for at se bort fra visse fysiske eller implementations-orienterede detaljer i forbindelse med et programs udførelse på et datamat-system. Havde man i stedet fuld viden om disse detaljer kunne man meget vel tænke sig at man kunne forudsige det præcise resultat. Denne tanke ligger også bag tidligt arbejde inden for nondeterminisme, f.eks. [Mil75].

I [4] betragtes et `while`-sprog hvor det antages at de nondeterministiske valg foretages ved at spørge et ukendt *orakel*, dvs. en uendelig sekvens af 0'er og 1'er. Der udvikles så en modificeret form for abstrakt fortolkning hvor nogle af komponenterne i funktionsargumenterne erstattes af approximativ information (beskrivende en mængde af værdier) mens andre komponenter stadig er enkelte værdier. I dette sprog erstattes orakler så af en approximativ information der beskriver et vilkårligt orakel, dvs. en vilkårlig sekvens af 0'er og 1'er. Det vises dernæst at den resulterende inducerede semantik er ækvivalent med den sædvanlige nondeterministiske denotationssemantik for dette sprog. — Så med et tilstrækkeligt stærkt potentsdomæne kan nondeterminisme altså formelt vises at være en anvendelse af abstrakt fortolkning hvad der allerede blev argumenteret uformelt for ovenfor.

Man kan så gå dybere ned i oraklernes struktur. Det viser sig således at for at beskrive den sædvanlige nondeterministiske denotationssemantik er det ikke nødvendigt at se på den over-tællelige mængde af alle orakler. Man kan nøjes med at se på en tællelig delmængde der dog skal indeholde de rekursive orakler (dvs. de orakler der kan listes af en Turing-maskine) som en *ægte* delmængde. Hvis de fysiske eller implementations-orienterede detaljer som man ser bort fra faktisk er beregnelige viser det at den sædvanlige nondeterministiske semantik specificerer ikke-terminering i for mange tilfælde. Dette problem optræder kun for den iterative konstruktion i sproget, dvs. for `while`-løkken, og [4] slutter af med at identificere en fix-punkt operator der specificerer ikke-terminering betydeligt sjældnere end den sædvanlige mindste fix-punkt operator. Der argumenteres så for at de to fix-punkt operatorer udgør mindste og største element i et fuldstændigt gitter af tilladelige fix-punkt operatorer. Endelig vises hvordan et operativ system vil kunne garantere at et program opfører sig som beskrevet af den nye fix-punkt operator.

3.3 TML: Oversættelsestidspunkt versus Kørselstidspunkt

Ingen af de to tilgangsvinkler til abstrakt fortolkning udvikler analyser tilstrækkeligt *automatisk* til at de umiddelbart er anvendelige i et værktøj til oversætterkonstruktion. De kan imidlertid bruges som et grundlag for at identificere de ingredienser som dette kræver.

Det er således nødvendigt at fokusere på et *metasprog* idet et oversætterkonstruktions-system skal behandle en (denotations[4]-)semantik på automatisk vis og metasproget netop har til formål at afgrænse de semantiske specifikationer der skal kunne håndteres. Den rolle som metasproget spiller her er altså ganske analog til den rolle som BNF spiller i specifikationen af programmeringssprog (dvs. som inddata til en parser-generator). Sædvanligvis er et metasprog for denotationssemantik baseret på en λ-kalkule med typer og lad os antage at der ikke er væsentligt andre konstruktioner i et sådant metasprog. Den væsentlige mangel ved dette metasprog er at det ikke giver tilstrækkelig fleksibilitet i fortolkningen af diverse konstruktioner. Det fremgår klart af de rekursive ligninger der betragtes i [3]: Betragt et program på formen

```
let f(x,y) = ...
and g(x,y) = ...
in f
```

hvor `x` er et heltals-argument, `y` er et sandhedsværdi-argument og resultaterne af funktionerne er heltal. Semantikken af dette program vil have formen

$$\text{FIX}(\ F\ ) \downarrow 1$$

hvor F er et semantisk funktional. Den detaljerede definition af F er uden betydning her men det er vigtigt at betragte funktionaliteten af F. Når vi definerer *standard semantikken*, dvs. beskriver uddata som en funktion af inddata, ville det være naturligt med

$$F \in (\mathbf{Z}\times\mathbf{B}\to\mathbf{Z}) \times (\mathbf{Z}\times\mathbf{B}\to\mathbf{Z}) \to (\mathbf{Z}\times\mathbf{B}\to\mathbf{Z}) \times (\mathbf{Z}\times\mathbf{B}\to\mathbf{Z})$$

hvor $\mathbf{Z}$ is et fladt domæne af heltal og $\mathbf{B}$ er et fladt domæne af sandhedsværdier. Når vi foretager en program-analyse som 'constant propagation' [ASU86] ville det være naturligt med

$$F \in (\mathbf{L}\times\mathbf{M}\to\mathbf{L}) \times (\mathbf{L}\times\mathbf{M}\to\mathbf{L}) \to (\mathbf{L}\times\mathbf{M}\to\mathbf{L}) \times (\mathbf{L}\times\mathbf{M}\to\mathbf{L})$$

hvor $\mathbf{L}$ er et fuldstændigt gitter ($\mathbf{Z}$ med et nyt største element) og $\mathbf{M}$ er et fuldstændigt gitter ($\mathbf{B}$ med et nyt største element). Det er en såkaldt *'independent attribute' analyse* fordi der ikke udtrykkes nogen sammenhæng mellem heltals- og sandheds-argumenterne. Hvis det i stedet blev formuleret som en såkaldt *relationel analyse* ville man kunne udtrykke sammenhænge mellem argumenterne og det ville da være naturligt med

[4]Valget af denotationssemantik vil blive motiveret i afsnit 4.

$$F \in (\mathrm{L}{\otimes}\mathrm{M}{\rightarrow}\mathrm{L}) \times (\mathrm{L}{\otimes}\mathrm{M}{\rightarrow}\mathrm{L}) \rightarrow (\mathrm{L}{\otimes}\mathrm{M}{\rightarrow}\mathrm{L}) \times (\mathrm{L}{\otimes}\mathrm{M}{\rightarrow}\mathrm{L})$$

for en passende operator $\otimes$. Begge disse analyser er *forlæns* analyser [ASU86]; ser vi dernæst på en *baglæns* analyse som f.eks. 'live variables analysis' ville det være naturligt med

$$F \in (\mathrm{N}{\times}\mathrm{N}{\leftarrow}\mathrm{N}) \times (\mathrm{N}{\times}\mathrm{N}{\leftarrow}\mathrm{N}) \rightarrow (\mathrm{N}{\times}\mathrm{N}{\leftarrow}\mathrm{N}) \times (\mathrm{N}{\times}\mathrm{N}{\leftarrow}\mathrm{N})$$

hvor N er et fuldstændigt gitter, f.eks. med elementerne `live` (dvs. at værdien kan blive brugt) og `dead` (dvs. at værdien ikke vil kunne blive brugt), og $\mathrm{N}{\times}\mathrm{N}{\leftarrow}\mathrm{N}$ er en forkortelse for $\mathrm{N}{\rightarrow}\mathrm{N}{\times}\mathrm{N}$.

Det er væsentligt at bemærke den måde på hvilken F's funktionalitet ændres. De basale typer, som f.eks. heltallene, kan således fortolkes som forskellige domæner og fuldstændige gitre, f.eks. **Z**, **L** og **N**. Også type-konstruktorerne bliver fortolket på forskellig vis afhængigt af om analysen er forlæns eller baglæns og om den er en 'independent attribute' analyse eller en relationel analyse. Der er imidlertid visse ingredienser i funktionaliteten der ikke ændres, nemlig funktionsrummet $\rightarrow$ der beskriver den beregning som F foretager og produkterne $\times$ der beskriver at argumentet til F og resultatet af F er et par af funktioner.

Det er altså nødvendigt med en version af λ-kalkulen hvor F kan få tildelt en tilstrækkelig fleksibel type til at ovenstående funktionaliteter kan fremkomme ved passende fortolkninger eller instantieringer. En mulighed er så

$$F \in \underline{\mathrm{Int}{\times}\mathrm{Bool}{\rightarrow}\mathrm{Int}} \times \underline{\mathrm{Int}{\times}\mathrm{Bool}{\rightarrow}\mathrm{Int}} \rightarrow \underline{\mathrm{Int}{\times}\mathrm{Bool}{\rightarrow}\mathrm{Int}} \times \underline{\mathrm{Int}{\times}\mathrm{Bool}{\rightarrow}\mathrm{Int}}$$

Det *væsentligste formål* med understregning er at kunne skelne mellem typer og typekonstruktorer der skal fortolkes eller instantieres forskelligt. Men det fremgår også at ikke-understregede typer og type-konstruktorer altid kan fortolkes på samme måde uafhængigt af hvilken analyse der foretages. Man kunne også tildele F en endnu mere fleksibel type som

$$F \in \underline{\mathrm{Int}{\times}\mathrm{Bool}{\rightarrow}\mathrm{Int}} \times \underline{\underline{\mathrm{Int}{\times}\mathrm{Bool}{\rightarrow}\mathrm{Int}}} \rightarrow \underline{\mathrm{Int}{\times}\mathrm{Bool}{\rightarrow}\mathrm{Int}} \times \underline{\underline{\mathrm{Int}{\times}\mathrm{Bool}{\rightarrow}\mathrm{Int}}}$$

Denne tillader forskellige fortolkninger af $\underline{\mathrm{Int}}$ og $\underline{\underline{\mathrm{Int}}}$ men fra et teoretisk synspunkt sker der ikke de store ændringer i den underliggende teori så der er ingen grund til at vælge denne mere komplekse notation.

Grundlaget for metasproget er et to-niveau type-system som netop illustreret. Den skelnen mellem typer som understregning giver anledning til nødvendiggør imidlertid en tilsvarende skelnen mellem udtryk. Vi kan derfor betragte en to-niveau λ-kalkule i hvilken typerne er som skitseret ovenfor og udtryk er som i λ-kalkulen men med den tilføjelse at visse konstruktioner kan være understreget. (Hertil kommer ydermere regler for hvornår udtryk er tilladelige, dvs. 'well-formed', og de er givet i artiklerne.) To-niveau λ-kalkulen gør det dog ikke tilstrækkeligt let at fortolke de understregede udtryk. Analogt til studiet af 'kategoriske kombinatorer' (når en λ-kalkule med typer fortolkes i en kartesisk lukket kategori) motiverer det indføjelsen af kombinatorer for de understregede konstruktioner i to-niveau λ-kalkulen. (Der er ingen grund til at gøre noget tilsvarende for de ikke-understregede konstruktioner da de jo betyder det samme uafhængigt af hvilken analyse der foretages.) Dermed er *to-niveau metasproget* TML blevet defineret og det er fundamentet for hovedparten af artiklerne. Som ved to-niveau λ-kalkulen er der visse regler som udtryk skal opfylde for at være tilladelige.

De regler som udtryk skal opfylde for at være tilladelige er dels af teknisk natur, dels er de forbundet med at de underliggende et-niveau typer skal stemme sammen og endelig er de en formalisering af den uformelle skelnen mellem oversættelsestidspunkt og kørselstidspunkt. (En undtagelse herfra er [JoNi91, afsnit 3] hvor jeg udvikler mindre restriktive regler.) Den fundamentale ide er at understregede konstruktioner, hvad enten det er typer eller udtryk, svarer til værdier eller beregninger på kørselstidspunktet, mens ikke-understregede konstruktioner svarer til værdier eller beregninger på oversættelsestidspunktet. Det betyder at en type der svarer til kørselstidspunktet ikke bør kunne indeholde typer der svarer til oversættelsestidspunktet da kørselstidspunktet jo kommer efter oversættelsestidspunktet. Forløbere for en formel skelnen mellem oversættelsestidspunkt og kørselstidspunkt er nævnt i nogle af artiklerne; her skal blot fremhæves Tennent's [Ten81] uformelle skelnen mellem *statiske udtryks-procedurer* og *udtryks-procedurer* og Paulson's [Pau84] *semantiske grammatikker.* — Da oversættelsestidspunkt og kørselstidspunkt er væsentlige begreber i oversætterkonstruktion kan man betragte brugen af to-niveau typer og udtryk som en måde at forbedre overensstemmelsen mellem (uformelle) *datalogiske* begreber og en udbredt (formel) *model* af typer og λ-udtryk.

Intuitionen bag den uformelle skelnen mellem oversættelsestidspunkt og kørselstidspunkt kan bruges som en hjælp når man vil indføre understregning i et almindeligt λ-udtryk (som det også fremgår af afsnit 3.6). Følger man denne tanke op er det naturligt at overveje om man kan udvikle en algoritme der automatisk indfører sådanne understregninger. Det vises i [13] at der findes en algoritme som givet

> et almindeligt λ-udtryk samt en delvist understreget type (svarende til værdier eller beregninger der skal foregå på kørselstidspunktet)

vil konstruere

> et *bedste to-niveau λ-udtryk*, dvs. et der opfylder betingelserne for at være tilladelig og hvor ikke mere end højst nødvendigt foregår på kørselstidspunktet.

(Dette formaliseres ved at definere passende partielle ordninger.) Denne algoritme til *bindingstids-analyse* er modelleret efter R.Milner's algoritme $\mathcal{W}$ [Mil78] for polymorf typeanalyse. En væsentlig forskel er imidlertid at der ikke i bindingstids-analysen er nogen analogi til den sammensætning af substitutioner der foregår i [Mil78] hvorfor algoritmen har en mere kompliceret rekursiv struktur. Korrekthedsbeviset er derfor ikke blot ved hjælp af strukturel induktion men ved hjælp af induktion over en mere kompliceret velordnet ('well-founded') ordning.

For så vidt angår kombinatorerne er det naturligt at tilpasse de eksisterende algoritmer for kombinator-introduktion, f.eks. [Sch24], [CuFe58], [Tur79] og [Hug82]. To-niveau strukturen giver visse komplikationer og [14] udvikler en transformation, kaldet *to-niveau λ-lifting*, der kan håndtere en delmængde af to-niveau λ-kalkulen. Den fulde to-niveau λ-kalkule håndteres i [16] men prisen er at der kræves ret omfattende transformationer og en kombinator, Ψ, der kan opfattes som ikke-implementerbar [NiNi89]. Fra et pragmatisk synspunkt synes det derfor bedre at restringere sig til en delmængde af to-niveau λ-kalkulen, f.eks. delmængden i [14]; denne kan evt. udvides med teknikkerne til undgåelse af Ψ som udvikles i [16] men det kræver endnu nogen forskning at kombinere disse. Der er derimod ingen problemer i at tilpasse bindingstids-analysen således at den konstruerer to-niveau λ-udtryk i den delmængde der betragtes i [14] (og det gøres f.eks. i [NiNi90, Kapitel 3]).

En uformel oversigt over bindingstids-analyse, to-niveau λ-lifting og meget af den efterfølgende udvikling gives i [NiNi88b]. Specielt beskrives hvorledes de enkelte ingredienser (fra afsnit 3.3 til 3.7) spiller sammen men udelukkende ved hjælp af eksempler.

3.4 Abstrakt Fortolkning baseret på Collecting Semantikken

I en række artikler udvikles abstrakt fortolkning for et delmængde af metasproget. Delmængden, TMLS, tillader ikke indlejrede understregede funktionstyper og det betyder i praksis at udviklingen kun omhandler PASCAL-lignende sprog. I dele af udviklingen er der behov for at typerne af de primitive funktioner opfylder endnu en betingelse, kaldet *'contravariantly pure'* (og som også blev brugt i [16] for at undgå Ψ). Det er lettest at forklare denne restriktion hvis man tænker på kodegenerering (se afsnit 3.6) hvor den umuliggør at en oversætter som hjælp for kodegenereringen foretager 'prøvekørsler' af kodestumper.

En kortfattet præsentation af denne udvikling gives i [5] og er baseret på [Nie84]. Semantikken af typer (herunder rekursive typer på begge niveauer) gives ved hjælp af den kategori-teoretiske tilgangsvinkel til domæne-teori [SmPl82]. For så vidt angår kørselstidspunktet vil der vise sig et behov for at kunne håndtere det såkaldte tensorprodukt og dette produkt er ikke en funktor over kategorien af algebraiske gitre (dvs. domæner der er fuldstændige gitre) og kontinuerte funktioner. I stedet for at restringere kategorien til en delkategori af såkaldt additive funktioner modificeres definitionen af funktor til et svagere begreb der kaldes semi-funktor. (Et tilsvarende begreb viser sig at have været studeret af kategoriteoretikere men det ligger uden for de fleste lærebøger i kategoriteori.) For så vidt angår oversættelsestidspunktet giver funktionsrum problemer da det jo er kontravariant i det ene argument. I stedet for at betragte en delkategori af såkaldte 'embeddings' (eller nedre adjungerede funktioner) følger [5] udviklingen i [Rey74] og betragter en kategori af domæner hvor morfierne er par af kontinuerte funktioner af formen $(f{:}D{\rightarrow}E, g{:}E{\rightarrow}D)$. Man kan så formulere funktionsrum som en covariant funktor over denne kategori.

Der defineres så en såkaldt 'eager' standard semantik for TMLS. (At den er 'eager' skyldes at der bruges strikte funktioner, 'smash'-produkt, 'coalesced' sum etc.). Dernæst defineres der en collecting semantik (eller *'lifted' standard semantik*[5]) men uden at inkludere programpunkter. Dette gør brug af det nedre potensdomæne (se ovenfor) og det vises at semantikken af en type der svarer til kørselstidspunktet er potensdomænet at den semantik som den samme type har i standard semantikken. Beviset er ved strukturel induktion på syntaksen for typer der svarer til kørselstidspunktet og grundet tilstedeværelsen af type variable (af hensyn til de rekursive typer) er der behov for en stærkere induktionsantagelse og denne formuleres ved hjælp af naturlige ækvivalenser. Dette resultat er den tekniske forklaring på hvorfor indlejrede understregede funktionsrum ikke tillades idet det ikke synes muligt at finde nogen operator $\boxed{\rightarrow}$ således at $\mathcal{P}_{\mathrm{H}}(D) \boxed{\rightarrow} \mathcal{P}_{\mathrm{H}}(E)$ er isormorf til $\mathcal{P}_{\mathrm{H}}(D{\rightarrow}E)$, hvor $\rightarrow$ står for strikt funktionsrum. Derudover viser det behovet for at betragte et tensorprodukt $\otimes$ der opfylder at $\mathcal{P}_{\mathrm{H}}(D)\otimes\mathcal{P}_{\mathrm{H}}(E)$ er isomorf med $\mathcal{P}_{\mathrm{H}}(D{\star}E)$, hvor $\star$ står for 'smash'-produkt.

Semantikken af udtryk gives ved at definere en afbildning af udtryk ind i elementer i de semantiske domæner og i [5] defineres både standard semantikken og collecting semantikken. For at vise korrekthed af collecting semantikken er det dernæst nødvendigt

[5]Jeg bruger udtrykket standard semantik selvom udelukkelsen af indlejrede understregede funktionsrum i det væsentlige restringerer semantikker til at være 'store' semantikker som i [1] og [2].

at definere relationer der udtrykker den ønskede sammenhæng. Det gøres ved induktion på strukturen af typer der svarer til oversættelsestidspunktet og der bruges her en teknik der kaldes logiske relationer i [Plo80] og relationelle funktorer i [Rey74]. Den ønskede sammenhæng vises så ved hjælp af et lemma der udsiger at semantikken af typerne og definitionen af relationerne sammen udgør en definition af en funktor over en passende kategori.

Dernæst udvikles abstrakt fortolkning. Første skridt er at formulere en ramme inden for hvilken man kan definere par af adjungerede funktioner (α_{rt},γ_{rt}) ved at foretage induktion på strukturen af typer (rt) der svarer til kørselstidspunktet. I forbindelse med type-konstruktorerne er der behov for at betragte naturlige transformationer mellem de semi-funktorer som type-konstruktorerne svarer til. De naturlige transformationer formaliserer forskellige måder at skifte mellem de analyse-metoder som de forskellige type-konstruktorer svarer til. Næste skridt er så at at definere logiske relationer ud fra disse par af adjungerede funktioner og at vise at hvis primitiverne i TMLs analyseres korrekt så analyseres alle udtryk i TMLs også korrekt.

Blandt de korrekte analyser kan man så søge at identificere 'mest præcise analyser' (kaldet *'best induced predicate transformers'* i [CoCo79]). Under antagelse af at de primitive funktioner alle har typer der opfylder betingelsen 'contravariantly pure' vises det at dette altid kan lade sig gøre; det går i det væsentlige ud på at udvide definitionen af de adjungerede par af funktioner (α_{rt},γ_{rt}) til adjungerede par af funktioner (α_{ct},γ_{ct}) der er indiceret af en type (ct) der svarer til oversættelsestidspunktet. Mere præcist kan man givet en analyse og et udvalg af approximative informationer definere en *induceret* analyse over disse. Den inducerede analyse er korrekt og er mindre approximativ end en vilkårlig anden korrekt analyse over det samme udvalg af approximative informationer. Da definitionen af (α_{ct},γ_{ct}) viser sig at være funktoriel i parametrene α og γ kan man udvikle inducerede analyser i mange små skridt og opnå samme resultat som hvis udviklingen var blevet foretaget i et stort skridt.

Fortolkningen af typer (rt) der svarer til kørselstidspunktet er kompositionel (dvs. foregår strukturelt) og derfor er fortolkningen af de understregede type-konstruktorer af særlig interesse. For understreget produkt er een fortolkning kartesisk produkt og som tidligere angivet svarer til en analyser i 'independent attribute form'. En anden mulighed er at bruge et tensor-produkt og givet karakterisationen af tensor-produkter på potensdomæner viser det at tensor-produktet, i hvert fald somme tider, svarer til relationelle analyser. Begrebet relationel analyse er imidlertid et ret intuitivt begreb og [6] giver derfor en karakterisation af tensor-produktet på en klasse af algebraiske gitre (der omfatter potensdomænerne) som kan tolkes som et generelt udsagn om at tensor-produktet formaliserer den relationelle metode. Derudover foretages en systematisk undersøgelse af hvorledes man kan skifte mellem forskellige analyse-metoder, dvs. forskellige naturlige transformationer defineres. — Den indsigt som dette giver er at væsentlige begrebsmæssige forskelle mellem formuleringer af program-analyser kan formaliseres som forskellige fortolkninger af understregede type-konstruktorer.

Eksistensen af inducerede analyser er teoretisk set meget interessant men de inducerede analyser er ikke nødvendigvis beregnelige. Så af beregneligheds-mæssige årsager og andre pragmatiske årsager er det umagen værd at betragte nogle af primitiverne i TMLs og undersøge forskellige 'forventede former' (*'expected forms'*) for deres definition. Disse forventede former bør være rimeligt lette at implementere og der bør være generelle korrekthedsresultater der en gang for alle viser det er korrekt at benytte dem ligesom der bør gives anvisninger på hvorledes forventede former skal ændres når analyse-metoderne ændres (ved at understregede type-konstruktorer fortolkes på anden vis). En sådan udvikling gives i [7] hvor der foreslås forventede former afhængigt af fortolkningen af de tilhørende understregede type-konstruktorer. Det vises så, ved at betragte de forskellige kombinationer af fortolkninger, at brugen af de forventede former altid vil være korrekt.

Der gives i [10] en oversigt over denne udvikling ligesom de fortagne valg yderligere motiveres. (Der medtages heri enkelte ider fra [9].) Specielt diskuteres anvendeligheden af potensdomæner og de problemer som konveksiteten (der forfindes i alle kendte potensdomæner) giver for collecting semantikken. Derudover illustreres det hvorledes programpunkter kunne tænkes anvendt i en analog udvikling af abstrakt fortolkning.

3.5 Abstrakt Fortolkning baseret på Standard Semantikken

Delmængden TMLS tillod ikke indlejrede understregede funktionsrum idet det ellers ikke var klart hvorledes collecting semantikken kunne defineres. Sammen med udviklingen i [BHA86] motiverer det en udvikling af abstrakt fortolkning hvor collecting semantikken spiller en mindre central rolle end ovenfor.

Et første skridt i den retning tages i [9]. Her betragter jeg en lille delmængde, TMLB, af to-niveau metasproget. I dette udelades rekursive typer af tekniske årsager idet jeg betragter monotone funktionsrum i stedet for kontinuerte funktionsrum. Endvidere beskrives der ikke nogen struktur for typer der svarer til kørselstidspunktet idet det jo var denne struktur (for så vidt angår understreget funktionsrum) der gav problemer i forbindelse med definitionen af collecting semantikken.

Syntaks og parametriseret semantik defineres i det væsentlige på samme måde som i foregående afsnit. *Korrekthed* formuleres dog nu som en relation mellem en analyse og standard semantikken og ikke som en relation mellem en analyse og collecting semantikken. Som et specialtilfælde omhandler det korrekthed af een analyse i forhold til en anden analyse (der f.eks. allerede er bevist korrekt) og dette specialtilfælde kaldes *sikkerhed* af een analyse i forhold til en anden. Næste trin er så at sikre eksistensen af *inducerede analyser*. Disse kan som ovenfor let specificeres ud fra en given analyse men da udviklingen i [9] netop ikke antager eksistensen af nogen analyse (som f.eks. collecting semantikken) er det ikke i sig selv nok.

Det er derfor nødvendigt at beskrive hvorledes en induceret analyse kan opnås ud fra standard semantikken. Det kan gøres ved at modificere den sædvanlige fremgangsmåde men der er visse problemer med at sikre at den derved beskrevne inducerede analyse faktisk eksisterer. Dette motiverer begrebet *'trofast mængde'* (*'faithful set'*) idet $\leq$ bliver til en partiel ordning på sådanne mængder. Det vises så at hvis man indskrænker sig til trofaste mængder så vil den inducerede analyse altid eksistere, være korrekt og så præcis som mulig. Som tidligere kan man så vise at inducerede analyser kan udvikles ved en 'trinvis forgrovning', hvor man gradvist betragter mere og mere approximative mængder af informationer, og det sikres at det giver samme resultat som hvis den inducerede analyse var induceret i eet skridt.

Det er imidlertid ret ubekvemt hele tiden at skulle restringere sig til trofaste delmængder. Således er det svært at give nogen generel karakterisation af de kompakte elementer hvorfor det er mere bekvemt at arbejde med monotont funktionsrum fremfor kontinuert funktionsrum. Der defineres derfor en syntaktisk restriktion, *'level-preserving'*, på typer. Den sikrer at semantikken af det tilhørende domæne i sig selv er en trofast mængde. Samtidig er det en generalisation af den syntaktiske restriktion 'contravariantly pure' som blev benyttet i afsnit 3.4. Endelig vises det at de fremgangsmåder der bruges i [MyJo86] og [BHA86] kan håndteres. — Da [BHA86] kun betragter strictness analyse og [MyJo86] ikke betragter inducerede analyser giver [9] et bedre indblik i graden af approximation som forskellige analyser udtrykker (herunder strictness analyse).

Det væsentlige formål bag [9] var at undgå at basere sig på eksistensen af collecting semantikken og det er derfor interessant at studere i hvilket omfang collecting semantikken faktisk eksisterer. I [9] vises det at collecting semantikken altid eksisterer hvis man restringerer sig til 'level-preserving' typer. Endvidere vises det at korrekthedsbeviser og inducering kan foretages ud fra collecting semantikken med samme resultat som hvis det foretages ud fra standard semantikken. (Det er her væsentligt at erindre at der i TMLB ikke beskrives nogen struktur af de typer der svarer til kørselstidspunktet.

Denne udvikling af abstrakt fortolkning skal så udvides til at omfatte et større metasprog end TMLB. Det gøres i [15] hvor generaliteten i udviklingen i afsnit 3.4 kombineres med den indsigt som [9] giver angående hvorledes man kan klare sig uden collecting semantikken. Udviklingen foretages for metasproget TMLM der i modsætning til TMLS tillader indlejrede understregede funktionsrum. I TMLM er der derfor basale typer, produkttyper, sum-typer, funktionsrum og rekursive typer på begge niveauer og TMLM svarer således til funktionalsprog snarere end blot PASCAL-lignende sprog. Endelig kan nævnes at den form for to-niveau λ-lifting der foretages i [14] giver udtryk i TMLM.

De væsentlige ideer bag udviklingen i [15] er:

- at koncentrere sig om typer der er *'level-preserving'* og ikke at tillade primitiver at have typer der ikke opfylder denne restriktion,
- at formulere konstruktionen af inducerede analyser ud fra Scotts partielle ordning, $\sqsubseteq$, fremfor den partielle ordning, $\leq$, der er af interesse i abstrakt fortolkning, og så
- efterfølgende vise at ovennævnte formulering er ækvivalent til en formulering ud fra $\leq$, dvs. at vise at den ovenfor beskrevne konstruktion har de ønskede egenskaber.

En mere udførlig oversigt over [15] følger. I afsnit 2 gives en gennemgang af forskellige tilgangsvinkler til abstrakt fortolkning. Metasproget, TMLM, defineres i afsnit 3 og i afsnit 4 udvikles dets parametriserede semantik. For typer der svarer til oversættelsestidspunktet følger det i det væsentlige [5]. For typer der svarer til kørselstidspunktet opnås øget generalitet ved at tillade indlejrede understregede funktionsrum og at tillade at semantikken a rekursive typer afhænger af fortolkningen. Det nødvendiggør en mere generel formulering hvor nu også semi-funktorerne (der svarer til typer fra kørselstidspunktet) formuleres over en kategori hvor morfierne er par af kontinuerte funktioner af formen $(f{:}D\rightarrow E, g{:}E\rightarrow D)$. Det parametriserede semamtik-begreb illustreres med to standard semantikker (hvoraf een er 'eager' og den anden er 'lazy') og med to analyser (fortegnsanalyse og 'liveness').

Afsnit 5 omhandler korrekthed af analyser i forhold til standard semantikken og, som et specialtilfælde heraf, sikkerhed af analyser i forhold til andre analyser. Det foregår i det væsentlige som i afsnit 3.4 ovenfor men er udvidet til at korretheds-relationerne nu også defineres strukturelt ud fra understregede typer (svarende til kørselstidspunktet). Ved hjælp af denne udvikling bevises fortegnsanalysen og 'liveness analysen' så korrekt (og det sidste ligger helt klart uden for [5]).

Afsnit 6 og 7 omhandler konstruktionen af inducerede analyser og behandler henholdsvis typer der svarer til oversættelsestidspunktet og typer der svarer til kørselstidspunktet. Begge afsnit begynder med det adjungerede tilfælde, hvor een analyse induceres ud fra en anden, hvorefter de betragter det generelle tilfælde, hvor en analyse induceres ud fra standard semantikken, og endelig undersøges eksistensen af collecting semantikken.

I afsnit 6 ignoreres strukturen af de typer der svarer til kørselstidspunktet og det antages derfor ikke at semantikken af disse typer er strukturelt defineret. Det bringer udviklingen ret tæt på [9] men en meget væsentlig forskel er at inducering kun betragtes for 'level-preserving' typer. Det bliver herved let at karakterisere de kompakte elementer i domænerne således at kontinuerte funktioner ikke volder problemer og det muliggør igen at rekursive typer kan klares. Beviserne bliver simplere fordi man kan udnytte definitionen af 'level-preserving' til at give relativt simple beviser for funktionsrum (i forhold til [9]). Som i [9] vises det så at inducerede analyser kan konstrueres ud fra standard semantikken og at man kan gøre det ved 'trinvis forgrovning' som beskrevet ovenfor. Endelig vises det at collecting semantikken altid eksisterer; det er her vigtigt at erindre at strukturen af de understregede typer netop ikke betragtes.

I afsnit 7 bruges naturlige transformationer til at skifte mellem forskellige fortolkninger af de understregede type-konstruktorer og et begreb der kaldes 'naturlig adjunktions-transformer' bruges til at skifte mellem forskellige fortolkninger af rekursive typer. (Det lykkedes mig her ikke at finde et egentligt kategorisk begreb da der kan være behov for at skifte fra en funktor over een kategori til en funktor over en anden kategori.) Der gives endvidere en række eksempler på hvorledes man kan skifte mellem fortolkninger af konkrete basale typer og type-konstruktorer (herunder fra 'independent attribute' metoder til relationelle metoder). Som i [5] er jeg ikke i stand til at give nogen definition af collecting semantikken; det skal her erindres at strukturen af de understregede typer jo netop betragtes hvilket giver de samme problemer som i [5] hvad understreget funktionsrum angår. — Den udvikling af abstrakt fortolkning der gives i [15] er derfor langt mere tilfredsstillende end udviklingen i [5] da der ikke længere er noget behov for at basere sig på collecting semantikken.

Afsnit 8 betragter så forventede former som i [7] men nu for hele TMLm. Afsnit 9 giver et sammendrag af udviklingen og det herunder fremhæves betydningen af de forskellige restriktioner der pålægges TMLm.

3.6 Kodegenerering

Program-analyser er på ingen måde den eneste ingrediens i konstruktionen af oversættere. Det er endnu vigtigere at kunne generere korekt kode og det ville derfor være hensigtsmæssigt om to-niveau metasproget kunne bruges som et hensigtsmæssigt udgangspunkt for en teori for korrekt kodegenerering på samme måde som det har været brugt som udgangspunkt for en teori for abstrakt fortolkning. Denne potentielle anvendelse har faktisk ligget bag definitionen af to-niveau metasproget lige fra begyndelsen (se f.eks. en bemærkning i [Nie84]) og det er værd at bemærke at når man ser bort fra denne anvendelse kan man formulere abstrakt fortolkning for et to-niveau metasprog der er mindre restriktivt end TMLm (hvilket jeg gør i [JoNi91, Afsnit 3]).

I [11] udvikles en teori for kodegenerering for metasproget TMLs. Afsnit 2 motiverer metasproget og afsnit 3 definerer dets parametriserede semantik ud fra samme tilgangsvinkel som er blevet brugt i artiklerne om abstrakt fortolkning. Den standard semantik der benyttes er 'eager', dvs. der bruges strikte funktioner, 'smash'-produkt og 'coalesced' sum.

Afsnit 4 defirer så en simpel abstrakt maskine, AM. Den er en simplificeret udgave af L. Cardelli's FAM [Car83] som er blevet brugt i forbindelse med implementationen af programmeringssproget ML. Da FAM er designet med henblik på ML, og da λ-kalkulen er indeholdt i ML, er det ikke overraskende at AM og FAM har mange træk fælles med den kategoriske abstrakte maskine, CAM [CCM87]. Dog bruger AM symbolske adresser og modellerer koden som en lineær sekvens af instruktioner.

Maskinens semantik gives ved hjælp af et transitionssystem der arbejder på konfigurationer bestående af en programtæller, en værdi-stak og en stak med returaddresser. Hver transition svarer til udførelse af en maskininstruktion. Det er ikke helt trivielt at beskrive sammenhængen mellem værdierne på værdi-stakkene og værdierne i semantikken for TMLs. Førstnævnte er idet væsentlige blot 'bitmønstre' mens sidstnævnte er elementer i semantiske domæner der kan være betydeligt mere komplicerede. Der defineres derfor *repræsentations-funktioner* der afbilder semantiske værdier ned i 'bitmønstrene'. For rekursive typer består det i at repræsentere et element ved dets endelige udfoldning. (Endeligheden garanteres af at standard semantikken er 'eager'.)

Afsnit 5 definerer en fortolkning der genererer kode til den abstrakte maskine. Kun fix-punkter giver anledning til problemer. Når det drejer sig om et fix-punkt for et understreget funktionsrum forventes kode hvor det rekursive kald foretages af en *call*-instruktion der stakker retur-addressen til senere brug. Gensidig rekursion kan f.eks. klares ved hjælp af Bekic's sætning der viser hvorledes gensidig rekursion kan udtrykkes som indlejret ikke-gensidig rekursion. En tilsvarende transformation kan bruges til at tillade summer af typer. For typer der ikke indeholder understregede symboler er der overhovedet ingen kodegenerering involveret så her vil det mindste fix-punkt være anvendeligt. (Det ville her være hensigtsmæssigt med teknikker der kan sikre at oversætteren ikke kan gå ind i en løkke.) Som det illustreres i [11] er det ikke umiddelbart muligt at tillade de resterende typer og der defineres derfor en syntaktisk restriktion, *'composite'*, på de typer som man kan specificere fix-punkt for.

Afsnit 6 beskæftiger sig med de mere pragmatiske konsekvenser af de restriktioner der er blevet indført i det forgående. Det vises at semantikken for SMALL [Gor79] systematisk kan transformeres til en form der tillades af [11]. Dette involverer fjernelse af 'continuations' (som beskrevet i [MiSt76]) og brugen af blok-numre og 'offsets' i stedet for adresser. Sidstnævnte teknik er af særlig interesse. Dels forekommer den også i [MiSt76] i forbindelse med udviklingen af en *'stack' semantik* fra en *standard semantik* men i modsætning til [MiSt76] er udviklingen her klart motiveret (som en måde at opfylde en restriktion der pålægges i [11]). Dels viser den hvorledes teknikker fra 'almindelig' oversætterkonstruktion kan finde anvendelse inden for *semantik-dirigeret oversætterkonstruktion*: Man starter med en semantik i en ret stor delmængde af to-niveau metasproget og den transformeres ved hjælp af forskellige heuristikker gradvist ned i en ret lille delmængde for hvilken korrekt og automatisk implementation kan foretages. — Denne sammenkædning af teknikker fra 'almindelig' oversætterkonstruktion og semantik-dirigeret oversætterkonstruktion er efter min mening ny.

I afsnit 7 vises så korrektheden. Det består af fire trin. I første trin defineres de ønskede korrektheds-relationer for understreget funktionsrum ved hjælp af de repræsentations-funktioner der blev defineret tidligere. Derudover defineres visse hjælpe-prædikater der f.eks. udtaler sig om at koden opfører sig 'pænt'. I andet trin udvides disse relationer til alle typer der svarer til oversættelsestidspunktet. Den basale ide er at bruge logiske relationer som for abstrakt fortolkning. I korrekthedsbeviset er der imidlertid behov for først at vise at koden opfører sig 'pænt' før de mere semantiske egenskaber inddrages. Dette muliggøres ved at anvende Kripke-lignende relationer og de er i det væsentlige blot logiske relationer der afhænger af en ekstra parameter (med en særlig anvendelse for funktionsrum). I tredje trin vises så at de forskellige primitiver oversættes korrekt. Det er mestensdels uden de store problemer men utroligt omstændeligt. For fix-punkter er det imidlertid nødvendigt at betragte en modificeret maskine der kan bringes til ikke at terminere når en vis rekursionsdybde nås. (Information herom indgår i 'parameteren' til de Kripke-lignende relationer.) I fjerde og sidste trin bruges det så til et strukturelt bevis af at der genereres korrekt kode for alle udtryk i det betragtede metasprog.

Ideelt set skulle ovennævnte udvikling have været foretaget for TMLM fremfor TMLS. Det er let nok at formulere kodegenereringen, se f.eks. [8] hvor 'eager' såvel som 'lazy' varianter betragtes, men definitionen af korrekthed følger ikke af [11] idet man ikke kan forvente at repræsentations-funktioner eksisterer for understregede funktionsrum. Det er derfor uklart hvorledes man kan bevise korrektheden af koden der genereres i [8] men man vil formentlig kunne anvende nogle af teknikkerne fra [Plo77].

3.7 Program-transformationer

Nogle af de transformationer der er blevet beskrevet i det foregående kan betragtes som program-transformationer. Det gælder f.eks. bindingtids-analyse og to-niveau λ-lifting og det gælder de transformationer på semantikken for SMALL der blev nævnt ovenfor. Mere traditionelle program-transformationer blev betragtet i afsnit 3.1 hvor resultater af program-analyser blev brugt til at validere program-transformationer der ikke under alle omstændigheder bevarede programmers mening.

I [12] bygges en bro mellem to-niveau type-strukturen og den klasse af program-transformationer der går under betegnelsen *partiel evaluering*. Partiel evaluering modelleres der ved hjælp af en operationel semantik, dvs. et reduktions-system der omfattter β-reduktion, men på en sådan måde at kun beregninger der svarer til oversættelsestids-punktet må foretages. I praksis vil man forvente at partiel evaluering altid terminerer idet en oversætter (og en to-niveau semantik) normalt konstrueres så dette sikres men som allerede nævnt har jeg ingen teknikker der kan sikre dette.

Uheldigvis synes der ikke at være enighed om hvad partiel evaluering egentlig er. Næppe nogen deltager ved den workshop hvor [12] blev præsenteret ville være uenig i at ovenstående er *en del af* partiel evaluering; men nogle af deltagerne ville tillade ganske anderledes kraftige program-transformationer f.eks. ved hjælp af teknikker fra automatisk bevisførelse. Når der ikke er fuld enighed om hvad partiel evaluering er kan der selvfølgelig heller ikke være fuld enighed om hvornår en partiel evaluator er bedre end en anden. Givet en definition af 'partiel evaluering' kan man imidlertid søge at formalisere de forskellige fordele som forskellige partielle evaluatorer har. Udviklingen i [12] var motiveret i den manglende formalisme[6] i mange artikler om partiel evaluering og af den opfattelse at man ikke kan give præcise argumenter før man har klargjort begreberne (f.eks. i form af definitioner). Dette er analogt til abstrakt fortolkning hvor man kun kunne argumentere intuitivt for en analyses korrekthed indtil korrekthed blev formaliseret ved hjælp af relationer som $\leq$.

Ud fra ovenstående opfattelse af hvad 'partiel evaluering' er, definerer [12] tre relationer der søger at formalisere hvornår en partiel evaluator er bedre end en anden. Fordele og ulemper ved disse relationer diskuteres dernæst. Derudover indeholder [12] den første præsentation af to-niveau λ-kalkulen (til forskel fra to-niveau metasproget) og indeholder den første bindingstids-analyse for en λ-kalkule med typer. I konklusionen diskuteres hvorledes man kan ændre 'well-formedness' betingelserne således at de bedre passer til andre[7] opfattelser af bindingstider, f.eks. som i [JSS85]. (Dette er analogt til, men ikke

[6]Ønskværdigheden af formalisme kan selvfølgelig diskuteres.

[7]Bemærk at $\Downarrow$ svarer til den kombinator Ψ der blev 'forkastet' i afsnit 3.3.

det samme som, de svækkelser af restriktioner der sker i f.eks. [JoNi91, Afsnit 3].)

I [18] betragtes en større klasse af program-transformationer end blot partiel evaluering. Udviklingen foretages for et lille eksempel-program og har ikke til hensigt at sikre at transformationerne kan anvendes på *automatisk vis*. Denne tilgangsvinkel tillader en samlet fremstilling af den måde på hvilken program-analyser kan interaktere med program-transformationer i forbindelse med at forbedre implementationers effektivitet.

De program-transformationer der illustreres i [18] kan groft inddeles i tre grupper. En af grupperne omhandler transformationer der har til hensigt at hjælpe bindingstids-analysen. Algoritmen der foretager bindingstids-analyse vil altid give et korrekt resultat men det kan være at resultatet foretager flere beregninger på kørselstidspunktet end egentligt ønskeligt. (Dette kan være nødvendiggjort af de 'well-formedness regler der er for to-niveau λ-kalkulen.) Dermed motiveres transformationer der omordner det oprindelige udtryk således at færre beregninger vil blive foretaget på kørselstidspunktet. Eksempler på sådanne transformationer er at ændre funktionaliteten af fix-punkt operatorer og 'currying'.

Den anden gruppe transformationer har til hensigt at introducere kombinatorer på en mere hensigtsmæssig måde end gjort i algoritmen for to-niveau λ-lifting. Det drejer sig om *algebraiske transformationer* af samme type som f.eks. udviklet for funktionalsproget FP og disse er tilstrækkeligt generelle til at gælde for alle fortolkninger (eller i det mindste alle interessante fortolkninger). Disse transformationer ændrer beregninger på kørselstidspunktet analogt til at partiel evaluering ændrer beregninger på oversættelsestidspunktet.

Den tredje gruppe transformationer består af transformationer der kræver forudgående program-analyser. Et eksempel er at erstatte et udtryk med et andet udtryk der beregner et konstant resultat. Et andet eksempel er at fjerne et del udtryk. For at kunne anvende sådanne program-transformationer er det vigtigt at vide noget om de omgivelser hvor de tænkes anvendt og denne information repræsenteres af program-analyse information tilknyttet de relevante programpunkter (som i afsnit 3.1).

3.8 Udvidelse af metasproget

To-niveau metasproget fremkommer ved at man tager udgangspunkt i en λ-kalkule med typer og så indfører en skelnen mellem bindingstider og kombinatorer for udtryk svarende til den ene bindingstid. Vil man udvide dette metasprog er det derfor hensigtsmæssigt først at udvide den underliggende λ-kalkule. Denne kunne udvides med generelle former for polymorfi og universelle typer, abstrakte datatyper og eksistentielle typer, Martin-Löf lignede produkter og summer, klasser og nedarvning samt et proces-begreb.

I [17] foreslås derfor at udvide en λ-kalkule med en af disse muligheder nemlig processer som første-klasses værdier. Det bygger på ideerne i bla. CSP [Hoa85] og CCS [Mil80] men disse tilgangsvinkler tillader ikke umiddelbart processer som første-klasses værdier. Nyere tilgangsvinkler der tillader dette er f.eks. [Tho89] der præsenterer en udvidelse af CCS så første-klasses processer tillades men det resulterende sprog har ingen typer. Det skulle imidlertid være klart fra de tidligere afsnit at udviklingen af abstrakt fortolkning og kodegenerering har udnyttet typerne i λ-kalkulen og det motiverer at typerne bør spille en væsentlig rolle i forbindelse med oversætterkonstruktion.

Som et første skridt i den retning beskriver [17] et sprog med et type-system hvor typerne bla. udtrykker de kommunikationsmuligheder som en proces har. Det er her nødvendigt at undlade at lade typerne søge at udtrykke i hvilken rækkefølge forskellige kommunikationer kan foregå da man ellers ikke kan håbe på et type-system der er statisk afgørligt. Der gives så inferens-regler for hvilke udtryk der har hvilken typer men der udvikles ikke algoritmer til type-analyse ligesom det heller ikke vises om principale typer eksisterer.

Den operationelle semantik ignorerer type-informationen i programmerne og er ikke meget forskellig fra den operationelle semantik for CCS og har elementer fra den operationelle semantik for λ-kalkulen. (Den fra funktionalsprog kendte skelnen mellem 'lazy' og 'eager' positioner søges anvendt til at udtrykke hvilken ekstern kommunikation der tillades i hvilke omgivelser.) Der vises så en sætning der siger at den operationelle semantik bevarer udtrykkenes typer. Det betyder at visse statiske fejl, som f.eks. at et par af værdier bliver anvendt som en funktion, ikke kan optræde for udtryk der er 'well-formed'. I traditionel terminologi udtrykkes et sådant resultat ofte som slogan'et *'well-typed programs don't go wrong'* [Mil78]. — Mig bekendt er [17] den første præsentation af et sprog hvor et generelt type-begreb udvikles for et sprog hvor processer tillades som 'første-klasses værdier'.

4 Metoder

I dette afsnit vil jeg diskutere nogle af de valg af metoder som ikke er blevet behandlet ovenfor.

Operationel Semantik versus Denotationssemantik

Da denne forskning startede (se afsnit 3.1) var der næppe nogen tvivl om at denotationssemantik var det mest hensigstmæssige semantiske udgangspunkt:

- Den *aksiomatiske tilgangsvinkel*, f.eks. Hoare logik for partiel korrekthed, var ikke nær så anvendt som denotationssemantik. Endvidere synes den indirekte logiske natur at give mindre hjælp til abstrakt fortolkning og kodegenerering end denotationssemantik.

- Den *operationelle tilgangsvinkel*, f.eks. SECD-maskinen, syntes meget afhængig af en konkret (abstrakt) maskine således at det ville blive vanskeligt at formulere abstrakt fortolkning og kodegenerering på et passende abstraktionsniveau.

- Den *algebraiske tilgangsvinkel*, f.eks. ADJ-gruppens, kunne opfattes som en variant af denotationssemantik hvor man blot gjorde mere ud af den homomorfe struktur som de semantiske afbildninger skulle have.

Siden da er der imidlertid kommet andre former for operationel semantik. I [Plo81] definerer Plotkin en såkaldt strukturel operationel semantik (*'Structural Operational Semantics'*). Her anvendes en transitionsrelation mellem konfigurationer der typisk består af en syntaktisk komponent (svarende til en tekstuel repræsentation af en 'continuation' i denotationssemantik) og en semantisk komponent (f.eks. en maskines lager). Da programmer kan terminere tillades også en type konfigurationer der ingen syntaktisk komponent har. En variant af denne tilgangsvinkel er blevet kendt under betegnelsen naturlig semantik [CDDK86] (*'Natural Semantics'*). Her relaterer transitionsrelationen altid en 'to-komponent' konfiguration til en konfiguration uden nogen syntaktisk komponent.

For de imperative sprog og funktionalsprog som jeg har betragtet er naturlig semantik et godt alternativ til denotationssemantik. Dette er ikke mindst klart for kodegenerering, som det f.eks. demonstreres i [CDDK86] og i [DaJe86] (som jeg var vejleder på), og skyldes at der ikke er behov for en kompliceret matematisk teori som ved denotationssemantik[8]. For så vidt angår abstrakt fortolkning er det mindre klart hvor hensigtsmæssig naturlig semantik er da abstrakt fortolkning naturligt involverer partielle ordninger og der således kunne blive behov for en ikke helt triviel indførelse af partielle ordninger i naturlig semantik. — Konklusionen må imidlertid blive at både denotationssemantik og naturlig semantik synes at være lovende tilgangsvinkler i forbindelse med semantik-baseret sprog-implementation.

Strukturel operationel semantik er mindre hensigtsmæssig end naturlig semantik så længe man betragter programmeringssprog som kan beskrives ved hjælp af naturlig semantik. Det skyldes at naturlig semantik er syntaks-dirigeret på samme homomorfe måde som denotationssemantik, mens strukturel operationel semantik er syntaks-dirigeret på en noget mere kompliceret måde. Hvis man imidlertid vil tilføje parallelle sprogkonstruktioner så er naturlig semantik for svag og denotationssemantik kræver anvendelsen af potensdomæner og 'resumptions'. Her kan strukturel operationel semantik give anledning til en simplere tilgangsvinkel men det kræver yderligere forskning at undersøge dette.

For så vidt angår denotationssemantik har jeg hovedsageligt anvendt kontinuerte funktioner samt cpo'er med et mindste element. Af tekniske årsager har jeg anvendt monotone funktioner i [3], [4] og [9] (og ikke-kontinuere og ikke-monotone funktioner i [1] og [2]). Brugen af monotone funktioner udelukker rekursive typer men kan til gengæld gøre *visse* beviser simplere. Jeg har ikke brugt cpo'er uden mindste element og partielle kontinuerte funktioner da man kan simulere dette ved hjælp af kontinuerte funktioner og cpo'er med mindste element.

Programmeringssprog-baseret versus Metasprog-baseret

Når man ønsker et system der automatisk kan konstruere oversættere bliver man nødt til at identificere et metasprog i hvilket semantikkerne beskrives. Dette metasprog vil formentlig have visse restriktioner og restriktionerne kan afhænge af det aspekt af oversætterkonstruktion som man ønsker automatiseret ud fra semantikken. Det er derfor vigtigt for brugeren af et sådant system at disse restriktioner relateres til begreber i programmeringssprog.

Eksempler på sådanne restriktioner findes i flere af artiklerne:

- den formelle restriktion kaldet *'contravariantly pure'* udelukker at en oversætter som hjælp for kodegenereringen foretager testkørsler af stumper af kode,
- delmængden TMLs af TML udelukker første-klasses procedurer og funktioner og svarer således til PASCAL-lignende sprog ligesom TMLm svarer til funktionalsprog, og
- den formelle restriktion *'composite'* kan opfattes som en forkærlighed for 'direct-style' semantikker fremfor 'continuation-style' semantikker.

[8]Dette var faktisk også en af grundene til at Plotkin udarbejdede [Plo81].

Dette at identificere delmængder af et metasprog, her TML, er formentligt den eneste måde på hvilken man kan strukturere problemstillingerne inden for semantik-dirigeret oversætterkonstruktion. Enhver delmængde er en præcis angivelse af nogle muligheder og mangler ved de tilhørende teknikker. Derved fokuseres på de teknikker der bør generaliseres og hvor man bør udvikle heuristikker der kan transformere en semantik således at de givne teknikker bliver anvendelige. Et eksempel på dette blev givet i afsnit 3.6.

Programpunkter versus Ingen Programpunkter

Bortset fra [1] og [2] har jeg ikke brugt programpunkter, dvs. resultaterne af en abstrakt fortolkning kendes kun ved programmets start og slutning og ikke ved indre programpunkter. Som beskrevet i konklusionen i [10] kan man forestille sig en analog udvikling hvor programpunkter anvendes. Dette er *nødvendigt* hvis man vil anvende resultaterne til at sikre program-analyser der ikke nødvendigvis bevarer semantikken, jvfr. [2] og [18].

Jeg udelod i sin tid programpunkter som et led i de simplifikationer af [1] og [2] der var nødvendige for at jeg kunne håndtere et metasprog fremfor blot eksempel-sprog. Til gengæld var det min opfattelse at det ville være ret let at introducere dem igen. Som det fremgår af [HuYo88] er dette imidlertid ikke tilfældet. Problemet er at for et funktionalsprog (i modsætning til et imperativt sprog [2]) er evalueringsrækkefølgen for udtryk ikke specificeret i den standard semantik man normalt tager udgangspunkt i. Hvis f.eks. venstre deludtryk for en operation ikke terminer kan man ikke umiddelbart afgøre hvilken information man skal tilordne højre deludtryk idet dette afhænger af evalueringsrækkefølgen. — Yderligere arbejde er derfor påkrævet og dette vil formentlig påvise en klar sammenhæng mellem evalueringsrækkefølger og måder at indøre programpunkter på.

Automatisk versus Systematisk

I afsnit 2 blev formålet med emneområdet beskrevet som udviklingen af system der *automatisk* kan konstruere en oversætter. Hovedideen bag abstrakt fortolkning er at 'lave fejl på den sikre side'; motiveret heri har jeg løbende valgt delproblemer inden for oversætterkonstruktion og identificeret delmængder af TML for hvilke en korrekt og automatisk konstruktion kan gives. Dette har givet anledning til et eksperimentelt system, *PSI-systemet* [Nie87b], hvor man kan eksperimentere med forskellige fortolkninger for metasproget. En 'front end', kaldet *DAOS* [AnCh87], gør det muligt at arbejde med denotationssemantikker.

En fordel ved denne tilgangsvinkel er at den peger direkte frem mod endemålet. En ulempe er at den samlede systematik måske ikke fremtræder så sammenhængende idet der stadig er aspekter af oversætterkonstruktion hvor jeg ikke har tilstrækkeligt generelle teknikker. Man kunne derfor hævde at der er så mange problemer inden for automatisk oversætterkonstruktion at man i stedet for at fokusere på det *automatiske* aspekt burde fokusere på det *systematiske* aspekt, hvor man blot beskriver metoder som kan være til hjælp i en manuel oversætterkonstruktion. Dette synspunkt har ligget til grund for [18] (i modsætning til f.eks. [NiNi88b]). — Det synes som om at store dele at litteraturen inden for feltet har taget sidstnævnte tilgangsvinkel i stedet for at lægge så megen vægt på det automatiske aspekt.

Semantik versus Symbolmanipulation

For mig er TML (eller en delmængde heraf) det fundamentale metasprog og det er derfor for TML at jeg formulerer fortolkninger og parametriserede semantikker. Det kan imidlertid være mere bekvemt at skrive denotationssemantikker i andre notationer og dette muliggøres af de i afsnit 3.3 beskrevne transformationer. Denne tilgangsvinkel er helt analog til f.eks. (Standard) ML hvor man interesserer sig for funktioner med typer men hvor man ønsker at blive fri for at skrive flere typer end højst nødvendigt. Dette muliggøres af en algoritme for type-analyse som kan indføre typerne i (mange[9]) udtryk uden eksplicitte typer.

Det væsentlige indhold bag afsnit 3.3 er derfor at man kan skrive TML-udtryk ved at skrive et λ-udtryk uden typer men idet man samtidig giver den overordnede to-niveau type. *Type-analysen* kan så indføre (et-niveau) typer i λ-udtrykket, hvorefter *bindingstids-analysen* indfører (to-niveau) typer og *to-niveau λ-lifting* indfører kombinatorer.

I denne måde at betragte tingene på opfattes type-analyse, bindingstids-analyse og to-niveau λ-lifting som symbolmanipulation. Man kan give type-analyse et semantisk indhold ved at vise at 'well-typed programs do not go wrong' (se ovenfor). For bindingstids-analyse kan man sigte mod at vise at semantikken af et to-niveau λ-udtryk er 'faithful' til sig selv (se [9]). (Tilgangsvinklen i [Mog89] har visse ligheder med denne ide.) For to-niveau λ-lifting kan man vise at det oprindelige to-niveau udtryk har samme semantik som det transformerede udtryk. — Selv om man viser sådanne resultater har man imidlertid ikke resultater for det parametriserede semantik-begreb som TML tilordnes; men kun for den standard fortolkning der svarer til semantikken af λ-udtryk uden typer. Jeg har derfor ikke fundet det tilstrækkeligt interessant at tilordne type-analyse, bindingstids-analyse og to-niveau λ-lifting et formelt semantisk fundament.

5 Sammenligning med litteraturen

Dette afsnit er struktureret omkring nogle centrale emneområder og inden for disse suppleres de sammenligninger med litteraturen der allerede er givet ovenfor. Mere detaljerede sammenlingninger findes i de indleverede artikler.

Oversætterkonstruktions-systemer

Det første system der automatisk kunne behandle en semantisk specifikation er formentlig P. Mosses' SIS system[Mos79][10]. Dette system accepterer denotationssemantikker i en særlig notation men er ikke egentligt et oversætterkonstruktions-system idet det ikke har nogen direkte mulighed for at generere kode. Ligeledes er der heller ikke faciliteter der understøtter program-analyser eller korrekthed af kodegenerering. Man kan imidlertid definere en semantik der afhænger af fortolkningen af visse primitiver og denne fortolkning kan gives på et senere tidspunkt. På den måde fås en form for parametriseret semantik meget lig den jeg har arbejdet med. Systemets pragmatiske anvendelighed bygger bla. på at en (ufuldstændig) semantik simplificeres ved hjælp af partiel evaluering efterhånden

[9]I princippet viser Scott's arbejde at ethvert λ-udtryk uden typer kan tilordnes en type når rekursive typer tillades. Imidlertid er der ingen garanti for at en given type-analyse algoritme [Mil78] kan klare sådanne udtryk.

[10]Udviklingen i [Nie79] blev foretaget ved hjælp af SIS.

som flere og flere primitiver fortolkes. Den skelnen som jeg opnår på en eksplicit måde (ved at bruge understregning) fremkommer altså her på en implicit måde.

Mange tilgangsvinkler til automatisk oversætter-konstruktion har baseret sig på attributgrammatikker men i disse tilgangsvinkler er det semantiske aspekt sjældent særligt fremtrædende. Som et middel til at specificere (ikke nødvendigvis korrekte) oversættere for programmeringssprog har brugen af attributgrammatikker resulteret i praktisk anvendelige oversættere. Et eksempel på et system er L. Paulson's system [Pau84] der bruger et metasprog (kaldet *semantiske grammatikker*) der er mere udtryktsfuldt end de fleste andre notationer for attributgrammatikker. Som beskrevet i [11] kan man opfatte to-niveau metasproget som en udvidelse af attributgrammatikker og semantiske grammatikker. Et andet eksempel på et attributgrammatik-baseret system er [Wil81] der bruger programanalyser til at aktivere program-transformationer.

En anden klasse af tilgangsvinkler har brugt kombinatorer svarende til operationerne i algebraisk semantik. En tidlig artikel inden for området er af ADJ-gruppens [TWW81] der tilpasser F.L. Morris's [Mor73] fremstilling til en lidt mere algebraisk tankegang. Mange senere artikler har benyttet sig af varianter af de kombinatorer som P. Mosses har udviklet under betegnelsen *'action semantics'* [Mos80]. Det gælder bla. for N.D. Jones og H. Christiansen's [ChJo82] og P. Lee og U. Pleban's [LePl86]. Korrekthed af kodegenereringen betragtes kun i nogle af disse artikler, især [TWW81] og [Mos80], men kun for ret små imperative sprog. Ved en kombinator-baseret tilgangsvinkel er det hensigtsmæssigt at valget af kombinatorer er ret fast idet man ellers ikke kan opbyge en 'meta-teori' der letter konkrete anvendelser. Dette er tilfældet for P. Mosses' *'action semantics'* men gælder ikke altid for tilgangsvinkler der benytter hans teknikker, f.eks. tillader [LePl86] at vælge kombinatorer svarende til produktionerne i en grammatik for sproget. (Som en konsekvens heraf indeholder [LePl86] da heller ingen korrekthedsovervejelser af betydning.)

En anden tilgangsvinkel bygger på anvendelsen af partiel evaluering og 'mixed computation', f.eks. [JSS85]. Ideen er her at en semantik for et programmeringssprog bliver givet i form af en fortolker. Når fortolkeren er blevet forsynet med et program kan den så specialiseres i stil med hvad der også skete i SIS-systemet. Man kan imidlertid ikke umiddelbart generere anden kode end for den maskine i hvilken fortolkeren er skrevet og man får således ikke et oversætterkonstruktions-system som beskrevet i afsnit 2.

Andre tilgangsvinkler til oversætterkonstruktions-systemer findes i [JoSc80], [Wan82], [Ras82] og [Set83] (se [11]).

Korrekthed af Kodegenerering

En af de tidligste, og mest imponerende, udviklinger af en korrekt implementation ved hjælp af kodegenerering blev givet af R. Milne og C. Strachey [MiSt76]. De behandler et meget stort sprog men udviklingen er ikke tilstrækkeligt systematisk til at andre har tilegnet sig deres teknikker. Denne udvikling står i kontrast til de algebraiske tilgangsvinkler, f.eks. [TWW81] og [Mos80], hvor der er er pæn struktur på beviserne men hvor til gengæld programmeringssprogene er ret simple. Som beskrevet i [11] var et af formålene med mit arbejde inden for kodegenerering at bruge to-niveau formalismen til at udvikle pæne og systematiske beviser for en større klasse af sprog end i de algebraiske tilgangsvinkler. En forskel i tilgangsvinkel er at jeg genererer kode fra en 'direct-style' semantik hvorimod [MiSt76] (og [Wan82]) genererer kode fra en 'continuation-style' semantik.

En anden lovende tilgangsvinkel til korrekt kodegenerering præsenteres i [CDDK86] og [Des86]. Her bruges naturlig semantik som det semantiske fundament og der gives et eksempel på hvorledes en delmængde af programmeringssproget ML kan oversættes til maskinkode for den kategoriske abstrakte maskine, CAM. En fordel ved denne tilgangsvinkel er at en maskine næsten altid er operationelt defineret således at der ikke som i min denotationssemantik-baserede tilgangsvinkel bliver behov for at bevise sammenhængen mellem en operationel semantik og en denotationssemantik. Det kunne derfor være interessant at undersøge om ideen med to niveauer kan indpasses i naturlig semantik på en sådan måde at man får en form for parametriseret transitionsrelation.

Abstrakt Fortolkning

Abstrakt fortolkning blev oprindeligt udviklet af P. Cousot og R. Cousot som en udvidelse af J.B. Kam og J.D. Ullman's 'monotone data flow analysis frameworks'. Dette bygger dels på *collecting* semantikken (som de kaldte statisk semantik) og dels på par af *adjungerede* funktioner der relaterer approximative informationer til mængder af værdier. I [Don78] gives et første forsøg på at formulere disse ideer inden for rammerne af denotationssemantik og dette motiverede [1] (og senere [5]) hvor flere af bestanddelene i abstrakt fortolkning blev indkorporeret. Det er også værd at bemærke B.K. Rosen's højniveau dataflow-analyser [Ros77,Ros80] men som i J.B. Kam og J.D. Ullman's tilgangsvinkel er der ikke mange semantiske overvejelser.

Som det udførligt fremgår af afsnit 2 af [15] er der mange måder at behandle abstrakt fortolkning på. I [BHA86] bruges en standard semantik i stedet for 'collecting' semantikken og abstraktionsfunktioner erstattes af (hvad jeg kalder) repræsentationsfunktioner[11]. I [MyJo86] blev der udelukkende anvendt relationer, i stedet for (adjungerede) funktioner, og [Myc87] fortsatte denne tilgangsvinkel. Det motiverede [9] hvor jeg først generaliserer restriktionen 'contravariantly pure' til den svagere restriktion 'level-preserving' og dernæst foretager en udvikling svarende til [5] for analyserne i [BHA86]. I [15] bringes de forskellige synspunkter endnu tættere på hinanden men som svar på slogan'et 'abstract interpretation = correctness' fra [Myc87] argumenteres der for slogan'et 'abstract interpretation = correctness + induced analyses + implementable analyses'.

Andre tilgangsvinkler til abstrakt fortolkning kan findes i f.eks. [AbH87] og [Hug88] og specialtilfældet strictness-analyse i f.eks. [Hug86], [Mau86] og [Dyb87]. — Det mest karakteristiske ved min tilgangsvinkel er at den er baseret på et metasprog (for at opnå generalitet) og at den har et stærkt og formelt semantisk indhold.

Bindingstider

En skelnen mellem oversættelses-tidspunkt og kørsels-tidspunkt går igen i de fleste artikler om semantik-dirigeret oversætterkonstruktion, f.eks. [App85], [Cli84], [Pau84] og [Ras82]. I [Ras82] præsenterer M.R. Raskovsky således et system der ved en stribe transformationer omskriver en traditionel denotationssemantik til en oversætter. Dette bygger på en skelnen mellem oversættelses-tidspunkt og kørsels-tidspunkt men denne skelnen er ikke udtrykt så klart som i to-niveau metasproget idet der bla. kun tillades et lille fast udvalg af domæner. Korrekthed af transformationerne betragtes ikke og der er heller ikke nogen klar angivelse af hvilke denotationssemantikker der kan håndteres.

Også U. Jørring og W. Scherlis's arbejde med 'staging transformations' [JøSc86] bygger på en uformel skelnen mellem trin ('stages') der i det væsentlige svarer til min skelnen

[11]Det er ikke det samme som de repræsentationsfunktioner der blev nævnt i afsnit 3.6.

mellem bindingstider. Disse 'staging transformations' har til formål at adskille forskellige beregningstrin fra hinanden som hvis man f.eks. ville samle alle beregninger på oversættelses-tidspunktet på yderste niveau således at der ikke fandtes nogen inde i beregninger på kørsels-tidspunktet. En lignende motivation forekommer i [MoWa88].

Bindingstids-analyse er også blevet formuleret af DIKU-gruppen i forbindelse med partiel evaluering og 'mixed computation' [JSS85]. Resultatet af bindingstids-analysen angives ved annotationer i programmet svarende til min brug af understregning. En væsentlig forskel er at mit arbejde ([12] og [13]) er baseret på en λ-kalkule med typer mens DIKU-gruppens oprindelige arbejde er baseret på første-ordens sprog uden typer. En anden forskel er at [JSS85] foretager bindingstids-analyse ved hjælp af abstrakt fortolkning hvorimod [12] og [13] gør det ved hjælp af en 'statisk' analyse på udtryk. En senere artikel fra DIKU-gruppen [Mog89] har dog reduceret forskellene.

6 Fremtidig forskning

En væsentlig udfordring ligger i at udvide den TML-baserede tilgangsvinkel til også at kunne håndtere parallelle sprog-konstruktioner. Dette vil være et af målene i et ESPRIT BRA projekt om *'Provably Correct Systems'* hvor der bla. skal konstrueres en oversætter fra en delmængde af OCCAM-2 til en transputer-lignende arkitektur. Semantikken af OCCAM-2 og transputeren vil blive udtrykt ved hjælp af strukturel operationel semantik og beviset for kodegenerering i [11] vil derfor skulle modificeres. Dette kan give anledning til ændringer i TML ligesom det skal overvejes om det ville være hensigtsmæssigt med en strukturel operationel semantik også for det niveau i TML der svarer til oversættelses-tidspunktet samt om man kan formulere en parametriseret strukturel operationel semantik der er lige så anvendelig som den parametriserede denotationssemantik der bruges i [11].

En personlig tak

Jeg har lært meget af mine vejledere i de forløbne år: Peter Mosses ([Nie79]), Neil Jones (specialet [Nie81]) og Gordon Plotkin (Ph.D.-afhandlingen [Nie84]). I de senere år har en bevilling fra Statens Naturvidenskabelige Forskningsråd været til stor støtte ligesom et seniorstipendium ved Dines Bjørner's gruppe på Danmarks Tekniske Højskole. Min kone (Hanne Riis Nielson) har været en uvurderlig faglig og personlig støtte.

References

[AbH87] S.Abramsky & C.Hankin: *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[AnCh87] K.B.Andersen & S.R.Christensen: DAOS - Konstruktion af et Dynamisk Automatisk Oversætter System, *Rapport IR 87-05, Aalborg Universitetscenter*, 1987.

[App85] A.W.Appel: Semantics-directed code generation, *Proc. 12th ACM Conf. on Principles of Programming Languages* 315–324, 1985.

[ASU86] A.V.Aho & R.Sethi & J.D.Ullman: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[BHA86] G.L.Burn & C.Hankin & S.Abramsky: Strictness analysis for higher-order functions, *Science of Computer Programming* **7** 249–278, 1986.

[Car83] L.Cardelli: The functional abstract machine, *Bell Labs. Technical Report TR-107*, 1983.

[CCM87] G.Cousineau & P.L.Curien & M.Mauny: The Categorical Abstract Machine, *Science of Computer Programming* 8 173–202, 1987.

[CDDK86] D.Clément & J.Despeyroux & T.Despeyroux & G.Kahn: A Simple Applicative Language: Mini-ML, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 1986.

[ChJo82] H.Christiansen & N.D.Jones: Control flow treatment in a simple semantics-directed compiler generator, *Formal Description of Programming Concepts II*, D.Bjørner (ed.), North-Holland, 1982.

[Cli84] W.Clinger: The SCHEME 311 Compiler. An exercise in denotational semantics, *Proc. ACM Conference on LISP and Functional Programming* 356–364, 1984.

[CoCo79] P.Cousot & R.Cousot: Systematic design of program analysis frameworks, *Proc. 6th ACM Symp. Principles Prog. Lang.* 269–282, 1979.

[DaJe86] M.Dam & F.Jensen: Compiler Generation from Relational Semantics, *Proceedings ESOP 1986*, Springer Lecture Notes in Computer Science **213** 1–29, 1986.

[CuFe58] H.B.Curry & R.Feys: *Combinatory Logic* vol. **1**, North-Holland, 1958.

[Des86] J.Despeyroux: Proof of Translation in Natural Semantics, *Symposium on Logic in Computer Science*, 1986.

[Don78] V. Donzeau-Gouge: Utilisation de la Sémantique dénotationelle pour l'étude d'interpretations non-standard, *IRIA Report 273*, 1978.

[Dyb85] P.Dybjer: Using domain algebras to prove the correctness of a compiler, *Proc. STACS 1985*, Springer Lecture Notes in Computer Science **182** 98–108, 1985.

[Dyb87] P.Dybjer: Inverse Image Analysis, *Proc. ICALP 87*, Springer Lecture Notes in Computer Science **267** 21–30, 1987.

[Ger75] S.L.Gerhart: Correctness-preserving program transformations, *Proceedings 2nd ACM Symposium on Principles of Programming Languages* 54–66, 1975.

[Gor79] M.J.C.Gordon: *The Denotational Description of Programming Languages, an Introduction*, (Springer, Berlin), 1979.

[HeAs80] M.C.B.Hennessy & E.A.Ashcroft: A Mathematical Semantics for a Nondeterministic Typed λ-Calculus, *Theoretical Computer Science* **11** 227–245, 1980.

[Hoa85] C.A.R.Hoare: *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hug82] J.Hughes: Supercombinators: a new Implementation Method for Applicative Languages, *Proc. of 1982 ACM Conf. on LISP and Functional Programming*, 1982.

[Hug86] J.Hughes: Strictness detection in non-flat domains, *Proc. Programs as Data Objects*, Springer Lecture Notes in Computer Science **217**, 1986.

[Hug88] J.Hughes: Backwards Analysis of Functional Programs, *Partial Evaluation and Mixed Computation*, D.Bjørner, A.P.Ershov and N.D.Jones (eds.), 187–208, North-Holland, 1988.

[HuYo88] P.Hudak J.Young: A Collecting Interpretation of Expressions (without Powerdomains), *Proceedings of the 15th ACM Symposium on Principles of Programming Languages* 107–118, 1988.

[JoNi91] N.D.Jones & F.Nielson: Abstract Interpretation, inviteret bidrag (under udarbejdelse) til *The Handbook of Logic in Computer Science*, North-Holland, 1991.

[JoSc80] N.D.Jones & D.A.Schmidt: Compiler Generation from Denotational Semantics, *Proc. Semantics Directed Compiler Generation*, Springer Lecture Notes in Computer Science **94** 70–93, 1980.

[JSS85] N.D.Jones & P.Sestoft & H.Søndergaard: An experiment in Partial Evaluation: The Generation of a Compiler Generator, *Proceedings of Rewriting Techniques and Applications*, Springer Lecture Notes in Computer Science **202**, 1985.

[JøSc86] U.Jørring & W.L.Scherlis: Compilers and Staging Transformations, *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986.

[KaUl77] J.B.Kam & J.D.Ullman: Monotone data flow analysis frameworks, *Acta Informatica* **7** 305–317, 1977.

[LePl86] P.Lee & U.Pleban: On the use of LISP in implementing denotational semantics, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* 233-248, 1986.

[Mau86] D.Maurer: Strictness computation using generalized λ-expressions, *Proc. Programs as Data Objects*, Springer Lecture Notes in Computer Science **217**, 1986.

[Mil75] R.Milner: Processes: A Mathematical Model of Computing Agents, *Logic Colloquium 1973* 157–174, North-Holland, 1975.

[Mil78] R.Milner: A theory of type polymorphism in programming, *J. Comput. System Sci.*, 1978.

[Mil80] R.Milner: *A Calculus of Communicating Systems*, Springer Lecture Notes in Computer Science **92**, 1980.

[MiSt76] R.Milne & C.Strachey: *A Theory of Programming Language Semantics*, Chapman and Hall, 1976.

[Mog89] T.Mogensen: Binding time analysis for higher order polymorphically typed languages, *TAPSOFT 1989*, Springer Lecture Notes in Computer Science, 1989.

[Mor73] F.L.Morris: Advice on structuring compilers and proving them correct, *Proc. ACM Conf. on Principles of Programming Languages* 144–152, 1973.

[Mos79] P.D.Mosses: SIS — Semantics Implementation System: reference manual and user guide, *Rapport, Aarhus Universitet*, 1979.

[Mos80] P.D.Mosses: A constructive approach to compiler correctness, *Proceedings ICALP 1980*, Springer Lecture Notes in Computer Science **85** 449–462, 1980.

[MoWa88] M.Montenyohl & M.Wand: Correct Flow Analysis in Continuation Semantics, *Proc. ACM Conference on Principles of Programming Languages*, 1988.

[Myc81] A.Mycroft: Abstract Interpretation and Optimising Transformations for Applicative Programs, *Ph.D.-thesis*, University of Edinburgh, 1981.

[Myc87] A.Mycroft: A Study on Abstract Interpretation and 'Validating Microcode Algebraically', *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin (eds.), 199–218, Ellis Horwood, 1987.

[MyJo86] A.Mycroft & N.D.Jones: A relational framework for abstract interpretation, *Programs as Data Objects*, Springer Lecture Notes in Computer Science **217**, 1986.

[Nie79] F.Nielson: Compiler Writing Using Denotational Semantics, *DAIMI TR-10*, Aarhus Universitet, 1979.

[Nie81] F.Nielson: Semantic Foundations of Data Flow Analysis, *speciale, DAIMI PB-131*, Aarhus Universitet, 1981.

[Nie84] F.Nielson: Abstract Interpretation Using Domain Theory, *Ph.D.-afhandling, CST-31-84*, University of Edinburgh, 1984.

[NiNi86a] F.Nielson & H.R.Nielson: Comments on Georgeff's 'Transformation and Reduction Strategies for Typed Lambda Expressions', *ACM Transactions on Programming Languages and Systems* 8 *3* 406–407, 1986.

[NiNi86b] F.Nielson & H.R.Nielson: Code Generation from Two-Level Denotational Meta-Languages, *Programs as Data Objects*, Springer Lecture Notes in Computer Science **217** 192–205, 1986.

[NiNi86c] H.R.Nielson & F.Nielson: Pragmatic Aspects of Two-Level Denotational Meta-Languages, *Proceedings ESOP 1986*, Springer Lecture Notes in Computer Science **213** 133–143, 1986.

[Nie86a] F.Nielson: A Bibliography on Abstract Interpretation, *ACM SIGPLAN Notices* **21** *5* 31–38, 1986 and *Bulletin of the EATCS* **28**, 1986.

[Nie86b] F.Nielson: Correctness of Code Generation from a Two-Level Meta-Language (Extended Abstract), *Proceedings ESOP 1986*, Springer Lecture Notes in Computer Science **213** 30–40, 1986.

[Nie87a] F.Nielson: Strictness Analysis and Denotational Abstract Interpretation (Extended Abstract), *ACM Conference on Principles of Programming Languages* 120–131, 1987.

[Nie87b] H.R.Nielson: The Core of the PSI-system, *Rapport IR 87–02, Aalborg Universitetscenter*, 1987.

[NiNi88a] H.R.Nielson & F.Nielson: Automatic Binding Time Analysis for a typed λ-Calculus (Extended Abstract), *ACM Conference on Principles of Programming Languages* 98–106, 1988.

[NiNi88b] F.Nielson & H.R.Nielson: The TML-approach to compiler-compilers, *Rapport ID-TR 1988-47*, Danmarks Tekniske Højskole, 1988.

[NiNi89] H.R. Nielson & F.Nielson: The Mixed λ-Calculus and Combinatory Logic (an overview), *Proceedings of the International Conference on Computing and Information*, North-Holland, 39–45, 1989.

[NiNi90] F.Nielson & H.R.Nielson: *Two-Level Functional Languages*, forelæsningsnoter.

[Pau84] Compiler generation from denotational semantics, *Methods and Tools for Compiler Construction*, B.Lorho (ed.), 219–250, Cambridge University Press, 1984.

[Plo77] G.D.Plotkin: LCF considered as a programming language, *Theoretical Computer Science* 5 *3*, 1977.

[Plo76] G.D.Plotkin: A Powerdomain Construction, *Siam J. Comput.* **5** *3* 452–487, 1976.

[Plo80] G.D.Plotkin: Lambda-definability in the full type hieararchy, *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, (J.P.Seldin and J.R.Hindley, eds.), Academic Press, 1980.

[Plo81] G.D.Plotkin: A Structural Approach to Operational Semantics, *Rapport DAIMI FN-19*, Aarhus Universitet, 1981.

[Plo82] G.D.Plotkin: A Powerdomain for Countable Nondeterminism, *Proceedings ICALP 1982*, Springer Lecture Notes in Computer Science **140** 418–428, 1982.

[Ras82] M.R.Raskovsky: Denotational semantics as a specification of code generators, *Proc. SIGPLAN 1982 Symp. on Compiler Construction* 230–244, 1982.

[Rey74] J.C.Reynolds: On the Relation between Direct and Continuation Semantics, *Procedings 2nd ICALP 1974*, Springer Lecture Notes in Computer Science **14** 141-156, 1974.

[Ros77] B.K.Rosen: High-level data flow analysis, *Communications of the ACM* **20** *10* 712–724, 1977.

[Ros80] B.K.Rosen: Monoids for rapid data flow analysis, *SIAM Journal of Computing* **9** 159–196, 1980.

[Sch24] M.Schoenfinkel: Über die Bausteine der mathematischen Logik, *Mathematische Annalen* **92**, 1924.

[Set83] R.Sethi: Control flow aspects of semantics directed compiling, *ACM TOPLAS* **5** *4* 554–595, 1983.

[SmPl82] M.B.Smyth & G.D.Plotkin: The category-theoretic solution of recursive domain equations, *SIAM J. Comput.* **11** *4*, 1982.

[Smy78] M.B.Smyth: Powerdomains, *J. Comput. System Sci.* **16** 23–36, 1978.

[Ten81] R.D.Tennent: *Principles of Programming Languages*, Prentice-Hall, 1981.

[Tho89] B.Thomsen: A Calculus of Higher Order Communicating Systems, *Proceedings of the 1989 ACM Conference on Principles of Programming Languages* 143–154, 1989.

[Tur79] D.A.Turner: Another algorithm for bracket abstraction, *The Journal of Symbolic Logic* 44, 1979.

[TWW81] J.W. Thatcher & E.G. Wagner & J.B. Wright: More on advice on structuring compilers and proving them correct, *Theoretical Computer Science* **15** 223–249, 1981.

[Wan82] M.Wand: Deriving target code as a representation of continuation semantics, *ACM TOPLAS* 4 *3* 496–517, 1982.

[Wil81] R.Wilhelm: Global Flow Analysis and Optimization in the MUG2 Compiler Generating System, *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D.Jones (eds.), 132–159, Prentice-Hall, 1981.

A Errata

Errata i [5]:

- Side 17: Linie 12 skulle være:

  be proved by structural induction on contravariantly pure types ct)

Errata i [8]:

- Side 251, venstre spalte: Indsæt følgende mellem linie 8 og 9:

  $|\ \mathrm{case}(e_1,\ldots,e_k)\ |\ \underline{\mathrm{in}}_j \qquad (\text{for } \underline{+})$

- Side 254, højre spalte: Linie 28 og 29 skulle være:

$$\begin{aligned}\underline{\mathrm{K}}_e(\mathrm{fix}_{ct+ct'}) = \lambda G.\ \mathrm{is}_1(G\bot) \rightarrow\ & \mathrm{in}_1(\underline{\mathrm{K}}_e(\mathrm{fix}_{ct})(\mathrm{out}_1 \circ G \circ \mathrm{in}_1)),\\ & \mathrm{in}_2(\underline{\mathrm{K}}_e(\mathrm{fix}_{ct'})(\mathrm{out}_2 \circ G \circ \mathrm{in}_2))\end{aligned}$$

Errata i [10]:

- Side 219: Linie 7: '[Con77a]' $\mapsto$ '[Cou77a]'.
- Side 219: Linie 10: '[HBP86]' $\mapsto$ '[Han86]'.
- Side 228: Figur 1: Insæt en pil fra 'Programming language' til 'metalanguage'.
- Side 238: Linie 8 fra neden: '$\downarrow$2' $\mapsto$ '$\downarrow$1'.

- Side 238: Linie 3 fra neden: '$\sqsubseteq$' $\mapsto$ '$\leq$'.

Errata i [11]:

- Side 80: Sidste linie: '(d)' $\mapsto$ '$(\bot)(r^u_{n_d}(d))$'.

Errata i [14]:

- Side 338: Linie 11 skulle være:

$$\varphi(\mathrm{tt},\mathrm{penv}) = \begin{cases} \mathrm{penv} & \text{if penv} \neq () \\ ((\mathrm{x_a},\mathrm{tt_1})) & \text{if penv} = () \text{ and tt} = \mathrm{tt_1} \underline{\rightarrow} \mathrm{tt_2} \end{cases}$$

Errata i [15]:

- Side 200: Linie 8: '$\times \cdots \times$' $\mapsto$ '$\otimes \cdots \otimes$'.

B Terminologi

I litteraturen bruges 'collecting semantikken' for to forskellige begreber. For at klargøre dette er det hensigtsmæssigt først at genkalde sig den terminologi der bruges i [MiSt76]: En *standard semantik* er en semantik der ikke indeholder implementations-orienterede detaljer, en *'store' semantik* er en semantik hvor det såkaldte 'environment' er en del af tilstandskomponenten og i en *'stack' semantik* opfører 'environment'-et sig som en symboltabel i en oversætter. Lad os nu kalde en semantik for *'lifted'* hvis den arbejder med mængder af værdier i stedet for værdier og lad os kalde en semantik for *'sticky'* hvis den ved hvert programpunkt husker mængden af værdier der når det. Denne skelnen mellem 'sticky' og ikke-'sticky' semantikker er ikke væsentlig for flowchart-modellen, idet programpunkter her svarer til grene og således er let tilgængelige, men er vigtig i den denotationssemantiske formulering.

Udviklingen af abstrakt fortolkning i [CoCo79] begynder med en standard (eller 'store') semantik for et flowchart. Dernæst betragtes en såkaldt *statisk semantik* og det er blot en '(sticky) lifted' standard (eller 'store') semantik. I den denotationssemantiske formulering givet i [1] og [2] vælges en tilsvarende tilgangsvinkel men her bruges en såkaldt *collecting semantik* som mellemtrin og det er blot en 'sticky' standard (eller 'store') semantik. Da denne brug af 'statisk semantik' er i strid med anvendelsen af 'statisk semantik' i distinktionen mellem statisk og dynamisk semantik er 'collecting semantik' også blevet brugt i betydningen '(sticky) lifted' standard (eller 'store') semantik, f.eks. i [3], [4], [5], [6], [7], [9], [10] og [15].