# Calculating Compilers

Een wetenschappelijke proeve op het gebied van de Wiskunde en Informatica

Proefschrift ter verkrijging van de graad van doctor aan de Katholieke Universiteit Nijmegen, volgens besluit van het college van decanen in het openbaar te verdedigen op dinsdag 25 februari 1992, des namiddags te 15.30 door

Henricus Johannes Maria Meijer

geboren op 18 april 1963 te Curaçao N.A.

druk: Krips Repro Meppel

| | |
|---|---|
| promotores: | Prof. C.H.A. Koster |
| | Prof. dr. J.-J.Ch. Meyer |

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Meijer, Henricus Johannes Maria

Calculating Compilers / Henricus Johannes Maria Meijer. -
[S.l. : s.n.] -Ill.
Proefschrift Nijmegen.
ISBN 90-9004673-9
Trefw.: compilers.

voor pa

# Contents

# Preface

*Life*
*A million things for you to learn*
*but one by one those pages turn*
*Like history itself*

Dave Stewart $^{\text{and}}_{\text{the}}$ Spiritual Cowboys

The systematic derivation of implementations from formal descriptions of programming languages has fascinated many researchers in computing science. Tennent [102] for example says

> A theory of semantics should contribute to systematic composition and verification of programs, especially compilers. Indeed, a general notation for semantic specification would permit the development of a true compiler-generator, just as BNF led to the development of parser generators.

In contrast to the consensus about formal definition of syntax by means of some variant of context free grammars, such as Backus-Naur Form [34], there is no agreement about a concise, readable and easily manipulatable notation for specifying the semantics of programming languages. As a solution we propose the Squiggol style of programming [67, 12] combined with the pragmatic aspects of Action Semantics [78]. This blend covers traditional techniques such as Initial Algebra Semantics [43] and Attribute (or Affix) Grammars [55, 56]. The language of update schemes [72] is used to give high level descriptions of low level abstract machines.

Chapter 2 discusses our modifications to make traditional Squiggol fit into the common denotational semantics framework of cpo's and continuous functions. Previous accounts [63] (implicitly) assumed to work in the category SET of total functions between sets of values. This has the disadvantage that finite and infinite types constitute different worlds and that arbitrary recursion is impossible. The price to be paid for working in CPO is that partiality of both functions and values becomes unavoidable. Only a tiny bit of the concepts and theory introduced in Chapter 2 is put into use in the rest of the thesis. It remains for future work to find out useful applications for the unused layabouts. Recently Paterson [82] has shown how anamorphisms can be used to give denotational semantics of term graphs.

As soon as we wish to give a denotational semantics of a language construct that is not already inherently present in the semantic meta-language, we are actually digging in the mud. The concept C in question has to be encoded in terms of primitives that are available. The best thing to hope for is that the coding process does not violate our intuition about C. Examples of concepts that are awkward to deal with in conventional denotational semantics are, in increasing order of troublesomeness: pointers, nondeterminism, and concurrency.

To specify complex pointer operations *update schemes* as devised by Meijer [72] (no relative of the author) seem to be a good choice. Update schemes will be presented in Chapter 3. Constructive proofs for the fact that an elegant notation for specifying pointer manipulations is no superfluous luxury are the tutorials by Aït-Kaci [2] and Maier and Warren [22]. In his thesis, Meijer gives a very intricate semantic description of update schemes. Chapter 3 presents a more abstract view. We introduce update schemes as a general computational model by setting them alongside Predicate/Transition nets [41]. An intermediate vision is given by Osborne [81]. Ultimately he wants to produce a compiler for (a restricted class of) update schemes.

After the first two theoretical chapters we turn our attention to more practical applications of the theory. In Chapter 4 a realistic compiler is derived for a simple imperative language. The resulting compiler is realistic; it generates three-address code, uses short circuit evaluation for boolean expressions and includes tail-call elimination. The main driving force in the calculations is the introduction of additional *continuation* arguments to make implicit control-flow explicit. This is especially clear in the development of an implementation for a backtrack language in Chapter 5. Starting from a semantics based on relations, four continuations are introduced culminating in (the control part) of Warrens Abstract Machine [110]. Chapter 1 provides a gentle introduction to our methods and shows how a compiler for a simple functional language may be calculated.

Although the practical results of the last two chapters are encouraging, there is still a lot of work to be done. There seem to be strong links with partial evaluation and techniques such as binding time analysis. The generic, variable free, definition of the fusion law fogs the distinction between compile-time and run-time objects. A full inductive proof gives a much cleaner separation between the two. Another aspect in favour of explicit induction proofs and bound variables occurs when dealing with attribute grammar-like semantic descriptions. Here there appears to be no way of avoiding the ritual steps of an inductive proof and the elimination of bound variables leads to an explosion of low-level combinator code to direct attributes to their correct destinations. These observations contradict with the tendency within the Squiggol community to avoid induction proofs and bound variables to get easier proofs and more elegant derivations. If the approach advocated in this thesis is to be applied to full scale programming languages, some form of machine assistance is imperative. The regular patterns of the proofs are promising in this respect.

## acknowledgements

Writing this thesis would not have been possible without the love and patience of Lilian and Frédérique, and the help and friendship of Ross Paterson and Maarten Fokkinga. Mark van den Brand and José van Beckhoven; your dinners are fabulous. The development of a Squiggol theory over cpo's was prompted by John-Jules Meyer, though the rôle of fixed point fusion has gradually disappeared in favour of fixed point induction. Kees Koster gave me the freedom to ride my hobbyhorses. The "wmc" and "mmc" clubs acted as sounding boards for many of the ideas in this thesis. Hugh Osborne kindly provided TEXnical and linguistic assistance. Last but not least I would like to mention the ex-members of the Phoenix project.

# Chapter 1

# More Advice on Proving a Compiler Correct: Improve a Correct Compiler

*Papa don't preach*
*I'm in trouble deep*
*Papa don't preach*
*I've been losing sleep*
*Cause I made up my mind*
*And I'm keeping my baby*

Madonna

One of the objectives of denotational semantics is to give a precise description of programming languages that can serve as a standard against which implementations can be verified, or preferably, from which correct compilers can be derived. We want to use a *calculational* approach to derive correct and efficient implementations for programming languages from their denotational descriptions. Experience [75] has shown that it is not a good idea to validate an implementation a posteriori, rather development and proof should proceed hand in hand. Besides the correctness aspect, a transformational approach can help to understand the relationships that may or may not exist between various implementations, or even suggest alternative methods. This introductory chapter provides a leisurely exposition on a calculational approach to semantics directed compiler generation that will be developed in the remainder of this thesis.

## 1.1 The Compiler Correctness Problem

Many people [51, 76, 65, 17, 101, 77, 33, 19] have suggested the use of algebraic means to tackle the compiler correctness problem. Given a source language $L$, a target language $T$, their respective semantics $m \in L \to M$, $a \in T \to U$ and a compiler $c$ from $L$ to $T$, one seeks an encoding $e$ of the source semantics into the target semantics such that the following diagram commutes.

$$\begin{tikzcd}
L \arrow[r, "c"] \arrow[d, "m"'] & T \arrow[d, "a"] \\
M \arrow[r, "e"'] & U
\end{tikzcd}$$

By enforcing an algebraic structure on the different domains and defining the respective functions as homomorphisms, initiality of $L$ ensures commutativity. A sufficient condition to prevent trivial solutions resulting from taking $U$ as a final algebra, is to require $e$ to be injective, i.e. $U$ must be a true implementation of $M$.

An F-algebra is a pair $(A, \varphi \in AF \to A)$ consisting of the *carrier* set $A$ and the operations $\varphi$ of *signature* F, for some suitable functor F. A *homomorphism* $h$ between F-algebras $(A, \varphi)$ and $(B, \psi)$ is a structure preserving map $h \in A \to B$ that replaces the operations $\varphi$ by $\psi$, formally, $h \circ \varphi = \psi \circ hF$, or as a diagram

$$\begin{tikzcd}
AF \arrow[r, "\varphi"] \arrow[d, "hF"'] & A \arrow[d, "h"] \\
BF \arrow[r, "\psi"'] & B
\end{tikzcd}$$

From the *initial* F-algebra $(L, in)$ there is a unique homomorphism to any other F-algebra $(A, \varphi)$. To stress the importance of this unique morphism it is called by the name "catamorphism" and written using so called banana-brackets $(\!| in := \varphi |\!)$, or $(\!| \varphi |\!)_F$.

The abstract syntax of a programming language $L$ is the initial F-algebra $(L, in)$ where $in$ is the set of constructors of the abstract syntax. Therefore we can define a denotational semantics $m \in L \to M$ for $L$ as the catamorphism $(\!| in := \varphi |\!)$ by imposing an F-algebraic structure $(M, \varphi)$ on the semantic domain $M$. In a traditional, oxford-style, denotational description this is achieved by encoding $\varphi$ using $\lambda$-abstraction and application. As argued by Mosses, Watt and others [78, 111, 60], such language descriptions have rather poor pragmatic qualities. It is hard to identify essential semantic concepts of the language being described, and the (automatic) generation of compilers is virtually impossible. Partial evaluation [97] is no solution; reducing a mess does not yield order. Besides this critique of traditional denotational semantics, it must be said that on the other hand compiler writers seem to ignore the existence of formal semantics completely, probably because they don't like greek letters. This behavior is inexcusable as it undermines any attempt to implement software that has been proven correct.

*Action Semantics* as developed by Mosses and Watt [111, 78] is an attempt to improve the readability and modularity of formal descriptions of programming languages. The semantic domain $M$ is cast into a G-algebra $(M, \alpha)$ where the set of *actions* $\alpha$ corresponds to the run-time concepts of the programming language in question. For an imperative language an action algebra includes primitive actions such as assignment and action combinators such as sequencing, looping and conditionals. The essence of writing a semantics now lies in extracting from the action algebra $(M, \alpha \in M\mathsf{G} \to M)$, a compile-time algebra $(M, T\alpha \in M\mathsf{F} \to M)$ of the same signature as the abstract syntax. Polymorphic functions $T \in \forall M.(M\mathsf{G} \to M) \to (M\mathsf{F} \to M)$ are called *transformers* by Fokkinga [37]; the run-time operations $\alpha$ are *transformed* into the compile-time actions $T\alpha$. Fokkinga uses transformers to formulate the notion of *laws* categorically. The ADJ-group calls operations $T\alpha$ *derived* operations.

Classical approaches such as Initial Algebra Semantics and Action Semantics have not been concerned with calculational issues. They only provided a framework to prove the correctness of a given compiler respectively to structure language definitions. We want to derive a new compiler by calculation not to prove a given one correct. Any semantics $m \in L \to M$ derived from an action G-algebra $(M, \alpha)$ can be factored into a static *compiler* $d \in L \to S$ and a dynamic $S$ interpreter $a \in S \to M$ such that the diagram

$$\begin{tikzcd}
L \arrow[rr, "d = \llparenthesis in := T\ IN \rrparenthesis"] \arrow[dr, "m = \llparenthesis in := T\alpha \rrparenthesis"'] & & S \arrow[dl, "a = \llparenthesis IN := \alpha \rrparenthesis"] \\
& M &
\end{tikzcd}$$

commutes, by simply letting $d$ map a program into a textual representation of its denotation. Hence $(S, IN)$ is the initial G-algebra of the semantic algebra underlying $M$. Sethi [98] dubs this representation 'concrete semantics'. The free theorem (see Chapter 2) for $T \in \forall M.(M\mathsf{G} \to M) \to (M\mathsf{F} \to M)$ is

$$f \circ \beta = \alpha \circ f\mathsf{G} \quad \Rightarrow \quad f \circ T\beta = T\alpha \circ f\mathsf{F} \qquad \text{(Transformer)}$$

From this law, commutativity of the above diagram is immediate.

$$\begin{array}{ll}
& \llparenthesis in := T\alpha \rrparenthesis = \llparenthesis IN := \alpha \rrparenthesis \circ \llparenthesis in := T\ IN \rrparenthesis \\
\Leftarrow & \quad \text{fusion law, see below} \\
& \llparenthesis IN := \alpha \rrparenthesis \circ T\ IN = T\alpha \circ \llparenthesis IN := \alpha \rrparenthesis \mathsf{F} \\
\Leftarrow & \quad \text{(Transformer)} \\
& \llparenthesis IN := \alpha \rrparenthesis \circ IN = \alpha \circ \llparenthesis IN := \alpha \rrparenthesis \mathsf{G} \\
\equiv & \quad \text{definition catamorphism} \\
& true
\end{array}$$

Factoring a semantics into a compile-time and a run-time part is a disciplined form of partial evaluation. The work done at compile-time is computing the residual value in the algebra $(S, IN)$ out of an element from $(S, T\ IN)$. In ordinary partial evaluation IN (and $\alpha$) are the unknowns in the factorization process.

Usually the compiler generated as described above will not produce very efficient code as too little work can be done statically. The remedy is to *improve* the already correct compiler $d$. Improving a compiler derived from $m \in L \to M$ means finding an (injective) implementation $e \in M \to U$ such that

$$\begin{tikzcd} & L \arrow[dl, "m"'] \arrow[dr, "n"] & \\ M \arrow[rr, "e"'] & & U \end{tikzcd}$$

commutes. Ideally, the new semantics $n$ is calculated from the composition of the old semantics $m$ and the improving transformation $e$. If $m$ is the catamorphism $(\!| in := T\alpha |\!)$ we may try to find a F-algebra $(U, R\beta)$ (with new action algebra $(U, \beta \in UH \to U)$) such that the refinement $e$ is a homomorphism between $(M, T\alpha)$ and $(U, R\beta)$, i.e. $e \circ T\alpha = R\beta \circ eF$. In that case we have (by initiality of L) that the new semantics $n$ is given by the catamorphism $(\!| in := R\beta |\!)$. No induction is needed.

$$\begin{tikzcd} & (L, in) \arrow[dl, "(\!| in := T\alpha |\!)"'] \arrow[dr, "(\!| in := R\beta |\!)"] & & & MF \arrow[r, "T\alpha"] \arrow[d, "eF"'] & M \arrow[d, "e"] \\ (M, T\alpha) \arrow[rr, "e"'] & & (U, R\beta) & \Leftarrow & UF \arrow[r, "R\beta"'] & U \end{tikzcd}$$

In many cases, unfortunately, the above method does not yield a satisfying underlying action algebra. Instead we want to determine a $R\beta$ such that $e \circ (\!| T\alpha |\!)\ l = (\!| R\beta |\!)\ l \circ e$. Now a full structural induction proof on $l$ seems unavoidable (for a more thorough discussion see Chapter 2). In this chapter we therefore use structural induction for all our proofs, even if it is, technically speaking, redundant.

A useful heuristic to obtain a implementation function $e$ is to add an extra argument to the semantics $m$, much akin to the ever popular accumulating argument strategy [14], such that this argument is available at compile-time. Shifting work from run-time to compile-time is essential to generate realistic code. A low-level implementation results when the run-time operations are transformed into an easily and efficiently implementable form exploiting structural properties of the domains. This process is called *defunctionalization* [91]. Typical examples are converting stores into actual memory and continuations into concrete machine code or linked lists. Arguments of operations can become global variables provided at any moment there is at most one 'active' copy of that argument around. This is the case when the argument is passed sequentially between operations, i.e., the semantics is single-threading [95]. Otherwise the argument should be copied explicitly to maintain referential transparency.

Based on the improved semantics $n$ we can again generate a compiler $c \in L \to T$ that solves the original compiler correctness diagram.

$$\begin{tikzcd} & L \arrow[dl, "m"'] \arrow[dr, "n"] & & & L \arrow[rr, "c"] \arrow[dr, "n"'] & & T \arrow[dl, "a"] & & L \arrow[r, "c"] \arrow[d, "m"'] & T \arrow[d, "a"] \\ M \arrow[rr, "e"'] & & U & + & & U & & = & M \arrow[r, "e"'] & U \end{tikzcd}$$

**High-Level Semantics**

An approach that is very close the one proposed here is *High-Level Semantics* as developed by Lee [60]. A High-Level semantic description consists of two levels. The *macro* semantics[1] $L \to (I \to S||POT)$ maps a program into a function that given compile-time objects from $I$ yields a pair consisting of the static semantics $S$, for example type correctness, and a term of the run-time action algebra POT. The *micro* semantics $POT \to M$ provides an interpretation specifying the dynamic semantics of a program. For a single POT several interpretations may be given. Lee makes no attempt to prove the correctness of a desired model (e.g. machine code) against an intended model (e.g. direct semantics).

Although it is a good idea to make an explicit *distinction* between compile-time and run-time, it is a bad thing to make an explicit and fixed *separation* between the two. The principal strategy to generate good code is to transfer as much work as possible from run-time to compile-time. Fixing the borderline between the two a priori make this impossible. Only when we have found an action algebra close enough to be implemented directly, the semantics may be changed into a compiler that yields a concrete representation of its dynamic semantics.

**Diacritical Convention** When making proofs about two semantic definitions we are frequently required to consider and to compare pairs of values, one from each definition, which are both called by the same name. The diacritical convention as proposed by Stoy [100] is a convenient and systematic way of distinguishing such values. All names from the one definition are given *acute* accents (´) while the names belonging to the other get *grave* accents (`). By convention acute accents are used for decorating the more ‘abstract’ semantics, while concrete, more to the ‘ground’ semantics get grave accents.

## 1.2 A simple functional language $\mathcal{F}$

The remainder of this chapter derives an implementation for a simple functional language $\mathcal{F}$ along the lines sketched in the introduction. First the syntax and the static semantics

[1]POT abbreviates “prefix-form operator term”

of $\mathcal{F}$ are introduced. An initial direct semantics is improved into a semantics where arguments are passed via an argument stack. The implementation of environments is optimized by introducing de Bruijn-indices. Finally an actual implementation is sketched using update schemes.

## 1.3 Syntax

The example source language consists of simply typed λ-expressions with integer constants. Adding more elaborate data types, pattern matching and other sugar posses no fundamental problems, but at the moment would only obscure the presentation needlessly.

$$\begin{array}{rcl} \tau, \sigma \in type & ::= & \mathbb{N} \mid type \to type \\ A, B, E, F \in expr & ::= & var^{type} \\ & \mid & const^{type} \\ & \mid & (\lambda var^{type}.expr)^{type} \\ & \mid & (expr\ expr)^{type} \end{array}$$

We let $x, y, z$ range over $var$, and $c$ over $const$.

## 1.4 Semantics

In order to define $\mathcal{F}$ completely, a *static semantics* and a *dynamic semantics* must be supplied. The static semantics should describe all context conditions, for example type checking. Though some aspects of static semantics can be realized by refining the abstract syntax of the language, this is not always the case. Context dependent properties, such as whether the same bound variable occurs with the same type, cannot be imposed by a context free grammar. Usual practice in denotational semantics is to consider context conditions as 'syntactic', and assume only syntactically correct programs. This gives the undesirable situation that context dependency falls between two stools; it cannot be described by the (context free) syntax, while it is ignored in the semantics. In our opinion static semantics form an essential part of a language definition and thus should be treated as a first class citizen.

### 1.4.1 Static semantics

The static semantics $\mathcal{M}[\![\_]\!] \in expr \to \mathbb{B}$ checks if expressions are well-typed. An example of a well-typed expression is

$$(\lambda x^{\sigma}.x^{\sigma})^{\sigma \to \sigma}$$

for all types $\sigma$. An ill-typed expression is

$$(3^{\mathbb{N}}\ x^{\sigma})^{\tau}$$

for all types $\sigma$ and $\tau$.

Type consistency is checked using an auxiliary function $\mathcal{E}[\![\_]\!] \in expr \to env \to type$. An environment $\eta$ is a mapping $var \to type$ that represents the context-condition that all occurrences of the same bound variable have the same type.

$$\begin{aligned}
\mathcal{E}[\![x^\sigma]\!]\ \eta &= \sigma,\ \text{if}\ \eta\ x = \sigma \\
\mathcal{E}[\![c^\sigma]\!]\ \eta &= \sigma,\ \text{if}\ \mathbb{N} = \sigma \\
\mathcal{E}[\![(\lambda x^\sigma.B)^\tau]\!]\ \eta &= \tau,\ \text{if}\ \tau = \sigma \to \mathcal{E}[\![B]\!]\ \eta[x := \sigma] \\
\mathcal{E}[\![(F\ A)^\sigma]\!]\ \eta &= \sigma,\ \text{if}\ \mathcal{E}[\![F]\!]\ \eta = (\mathcal{E}[\![A]\!]\ \eta) \to \sigma
\end{aligned}$$

Using $\mathcal{E}[\![\_]\!]$ we can easily define $\mathcal{M}[\![E^\sigma]\!] = (\sigma = \mathcal{E}[\![E^\sigma]\!]\ \eta_0)$ where $\eta_0$ is the empty environment. The given static semantics is peculiar in that it is only a semi-decision procedure. For well-typed expressions $\mathcal{M}[\![\_]\!]$ yields true, but it is undefined for ill-typed expressions.

### 1.4.2 Standard semantics

The standard, call-by-name, interpretation of expressions is given by the valuation function $\mathcal{M}[\![\_]\!] \in expr \to D$, where $D$ is the solution of the recursive domain equation

$$D ::= \mathit{Value}\ \mathbb{N} \mid \mathit{Func}\ D \to D$$

Again $\mathcal{M}[\![\_]\!]$ is defined in terms of a helper function $\mathcal{E}[\![\_]\!]$, this time of type $expr \to env \to D$. An environment $\eta$ is a mapping $var \to D$ that records the values of the free variables that occur within an expression. The action algebra of actions $env \to D$ has as operations

$$\begin{aligned}
\mathsf{LOOKUP}\ x^\sigma\ \eta &= \eta\ x^\sigma \\
\mathsf{CONST}\ c\ \eta &= \mathit{Value}\ c \\
\mathsf{LAMBDA}\ x^\sigma\ e\ \eta &= \mathit{Func}\ (\lambda a.(e\ \eta[x^\sigma := a])) \\
\mathsf{APPLY}\ f\ a\ \eta &= (\mathit{Func}^{-1}\ (f\ \eta))\ (a\ \eta)
\end{aligned}$$

Call-by-name, or normal order evaluation is implied by the strictness of the destructor $\mathit{Func}^{-1}$ which forces evaluation of its argument $f$.

Given the above actions the definition of the valuation function $\mathcal{E}[\![\_]\!]$ is immediate.

$$\begin{aligned}
\mathcal{E}[\![x^\sigma]\!] &= \mathsf{LOOKUP}\ x^\sigma \\
\mathcal{E}[\![c^{\mathbb{N}}]\!] &= \mathsf{CONST}\ c \\
\mathcal{E}[\![(\lambda x^\sigma.E)^\tau]\!] &= \mathsf{LAMBDA}\ x^\sigma\ \mathcal{E}[\![E^{\sigma\to\tau}]\!] \\
\mathcal{E}[\![(F\ A)^\tau]\!] &= \mathsf{APPLY}\ \mathcal{E}[\![F^{\sigma\to\tau}]\!]\ \mathcal{E}[\![A^\sigma]\!]
\end{aligned}$$

Note that $\mathcal{E}[\![\_]\!]$ corresponds to a catamorphism

$$(\!|\mathsf{LOOKUP}, \mathsf{CONST}, \mathsf{LAMBDA}, \mathsf{APPLY}|\!)$$

that replaces the (invisible) constructor for variables by LOOKUP, that for constants by CONST etc.

We are only interested to compute the result of expressions of base type, so $\mathcal{M}[\![E^{\mathbb{N}}]\!] = \mathcal{E}[\![E^{\mathbb{N}}]\!]\ \eta_0$ where $\eta_0$ is the empty environment.

The typed λ-calculus admits a much simpler semantic domain, namely interpret $\mathbb{N}$ as the cpo of natural numbers and $\to$ as the continuous function space functor [85].

$$\begin{aligned} D_{\mathbb{N}} &= \mathbb{N} \\ D_{\sigma\to\tau} &= D_\sigma \to D_\tau \end{aligned}$$

Then we have that $\mathcal{E}[\![E^\sigma]\!] \in env \to D_\sigma$, where $env$ is the set of type respecting mappings from variables of type $\sigma$ to elements of $D_\sigma$.

### 1.4.3 Spineless semantics

A compiler based on the above semantics will probably result in an inefficient G-machine-like implementation [50]. Better results can be expected when an explicit argument stack is introduced. This will lead to implementations that are generalizations of the Spineless Tagless G-Machine [88] and the Three Instruction Machine [71]. The semantic domains for a spineless semantics are

$$\begin{aligned} \eta \in env &= var \to func \\ a, b \in func &= stack \to \mathbb{N} \\ \xi \in stack &::= [\,] \mid func \rightarrowtail stack \end{aligned}$$

These domain equations are again too loose, and well-typedness allows for more refined domains.

$$\begin{aligned} \eta \in env &= \{var^\sigma \to func_\sigma\} \\ [\,] \in stack_{\mathbb{N}} &= \mathbf{1} \\ a \rightarrowtail \xi \in stack_{\sigma\to\tau} &= func_\sigma \| stack_\tau \\ a \in func_\sigma &= stack_\sigma \to \mathbb{N} \end{aligned}$$

In order to transform the standard semantics into a spineless one, we define two mutual recursive binary operations, the concretization map $\check{}\ \in D\|type \to func$ and the abstraction map $\hat{}\ \in func\|type \to D$ such that for all $\sigma \in type$ and $\grave{f} \in func_\sigma$ and $\acute{f} \in D_\sigma$ the adjunctive property

$$\grave{f} = \acute{f}^{\vee\sigma} \quad \equiv \quad \acute{f} = \grave{f}^{\wedge\sigma} \qquad \text{(AbstrConcrIso)}$$

holds. Thus $\_^{\vee\sigma}$ and $\_^{\wedge\sigma}$ are each others inverses, when applied to arguments of the right type. The respective types of $\check{}$ and $\hat{}$ leave little choice.

$$\begin{aligned} c^{\vee\mathbb{N}}\ [\,] &= c & c^{\wedge\mathbb{N}} &= c\ [\,] \\ f^{\vee\sigma\to\tau}\ (a \rightarrowtail \xi) &= (f\ a^{\wedge\sigma})^{\vee\tau}\ \xi & f^{\wedge\sigma\to\tau}\ a &= (\lambda\xi.f\ (a^{\vee\sigma} \rightarrowtail \xi))^{\wedge\tau} \end{aligned}$$

The last two definitions may be written into variable free form using $(f \to g)\ h = g \circ h \circ f$.

$$\begin{aligned}
\_^{\check{}\sigma\to\tau} &= \mathrm{pop} \circ (\hat{}^{\sigma} \to \check{}^{\tau}) \\
\mathrm{pop}\ f\ (a \succ\!\!+ \xi) &= f\ a\ \xi \\
\_^{\hat{}\sigma\to\tau} &= (\check{}^{\sigma} \to \hat{}^{\tau}) \circ \mathrm{push} \\
\mathrm{push}\ f\ a\ \xi &= f\ (a \succ\!\!+ \xi)
\end{aligned}$$

Using abstraction and concretization operations we derive a spineless implementation.

$$\begin{array}{ll}
 & \grave{\mathcal{M}}[\![E^{\mathbb{N}}]\!] \\
= & \quad \text{demand} \\
 & \acute{\mathcal{M}}[\![E^{\mathbb{N}}]\!] \\
= & \quad \text{unfold} \\
 & \acute{\mathcal{E}}[\![E^{\mathbb{N}}]\!]\ \eta_0 \\
= & \quad \text{(AbstrConcrIso)} \\
 & (\acute{\mathcal{E}}[\![E^{\mathbb{N}}]\!]\ \eta_0)^{\check{}\mathbb{N}\,\hat{}\mathbb{N}} \\
= & \quad \text{assume theorem (1.1) holds} \\
 & (\grave{\mathcal{E}}[\![E^{\mathbb{N}}]\!]\ \check{\eta})^{\hat{}\mathbb{N}} \\
= & \quad \text{unfold} \\
 & \grave{\mathcal{E}}[\![E^{\mathbb{N}}]\!]\ \check{\eta}\ [\,]
\end{array}$$

In the above calculation we have assumed that

$$\_^{\check{}\sigma} \circ \acute{\mathcal{E}}[\![E^{\sigma}]\!] \quad = \quad \grave{\mathcal{E}}[\![E^{\sigma}]\!] \circ \check{\_} \tag{1.1}$$

where $\check{\eta}\ x^{\sigma} = (\eta\ x^{\sigma})^{\check{}\sigma}$ is the extension of the concretisation function $\check{}$ to environments. This theorem will be proved by structural induction. A new semantics $\grave{\mathcal{E}}[\![\_]\!]$ and a set of new run-time actions are determined as a side-effect of the proof. When giving hints within a calculation we use "fold" and "unfold" when compile-time values are manipulated and "evaluate" in case run-time expressions are simplified. New run-time operations are "synthesized" while new compile-time operations are "extracted".

The base case for the proof is $E := x^{\sigma}$.

$$\begin{array}{ll}
 & \_^{\check{}\sigma} \circ \acute{\mathcal{E}}[\![x^{\sigma}]\!] \\
= & \quad \text{unfold} \\
 & \_^{\check{}\sigma} \circ \mathrm{LOOKUP}\ x^{\sigma} \\
= & \quad \text{abutting calculation} \\
 & \mathrm{ENTER}\ x^{\sigma} \circ \check{\_} \\
= & \quad \text{extract} \\
 & \grave{\mathcal{E}}[\![x^{\sigma}]\!] \circ \check{\_}
\end{array}
\qquad\qquad
\begin{array}{ll}
 & (\mathrm{LOOKUP}\ x^{\sigma}\ \eta)^{\check{}\sigma}\ \xi \\
= & \quad \text{evaluate} \\
 & (\eta\ x^{\sigma})^{\check{}\sigma}\ \xi \\
= & \quad \text{fold} \\
 & \check{\eta}\ x^{\sigma}\ \xi \\
= & \quad \text{synthesize} \\
 & \mathrm{ENTER}\ x^{\sigma}\ \check{\eta}\ \xi
\end{array}$$

The next base case deals with constants.

$$\begin{aligned}
& \_^{\vee\mathbb{N}} \circ \acute{\mathcal{E}}[\![c^{\mathbb{N}}]\!] \\
= & \quad \text{unfold} \\
& \_^{\vee\mathbb{N}} \circ \mathrm{CONST}\ c^{\mathbb{N}} \\
= & \quad \text{abutting calculation} \\
& \mathrm{RETURN}\ c^{\mathbb{N}} \circ \check{\_} \\
= & \quad \text{extract} \\
& \grave{\mathcal{E}}[\![c^{\mathbb{N}}]\!] \circ \check{\_}
\end{aligned}
\qquad\qquad
\begin{aligned}
& (\mathrm{CONST}\ c^{\mathbb{N}}\ \eta)^{\vee\mathbb{N}}\ [\,] \\
= & \quad \text{unfold} \\
& \mathrm{CONST}\ c^{\mathbb{N}}\ \eta \\
= & \quad \text{evaluate} \\
& c \\
= & \quad \text{synthesize} \\
& \mathrm{RETURN}\ c^{\mathbb{N}}\ \check{\eta}\ [\,]
\end{aligned}$$

Abstractions need somewhat more work.

$$\begin{aligned}
& \_^{\vee\sigma\to\tau} \circ \acute{\mathcal{E}}[\![(\lambda x^{\sigma}.E)^{\sigma\to\tau}]\!] \\
= & \quad \text{unfold} \\
& \_^{\vee\sigma\to\tau} \circ \mathrm{LAMBDA}\ x^{\sigma}\ \acute{\mathcal{E}}[\![E^{\tau}]\!] \\
= & \quad \text{abutting calculation} \\
& \mathrm{TAKE}\ x^{\sigma}\ \grave{\mathcal{E}}[\![E^{\tau}]\!] \circ \check{\_} \\
= & \quad \text{extract} \\
& \grave{\mathcal{E}}[\![(\lambda x^{\sigma}.E)^{\sigma\to\tau}]\!] \circ \check{\_}
\end{aligned}
\qquad\qquad
\begin{aligned}
& (\mathrm{LAMBDA}\ x^{\sigma}\ \acute{\mathcal{E}}[\![E^{\tau}]\!]\ \eta)^{\vee\sigma\to\tau}\ (a \succ\!+ \xi) \\
= & \quad \text{unfold} \\
& (\mathrm{LAMBDA}\ x^{\sigma}\ \acute{\mathcal{E}}[\![E^{\tau}]\!]\ \eta\ a^{\wedge\sigma})^{\vee\tau}\ \xi \\
= & \quad \text{evaluate} \\
& (\acute{\mathcal{E}}[\![E^{\tau}]\!]\ \eta[x^{\sigma} := a^{\wedge\sigma}])^{\vee\tau}\ \xi \\
= & \quad \text{IH} \\
& \grave{\mathcal{E}}[\![E^{\tau}]\!]\ \check{\eta}[x^{\sigma} := a^{\wedge\sigma\vee\sigma}] \\
= & \quad {}^{\wedge\sigma\vee\sigma} = id \\
& \grave{\mathcal{E}}[\![E^{\tau}]\!]\ \check{\eta}[x^{\sigma} := a]\ \xi \\
= & \quad \text{synthesize} \\
& \mathrm{TAKE}\ x^{\sigma}\ \grave{\mathcal{E}}[\![E^{\tau}]\!]\ \check{\eta}\ (a \succ\!+ \xi)
\end{aligned}$$

Finally for applications we need some creativity to reach the induction step.

$$\begin{aligned}
& \_^{\vee\sigma} \circ \acute{\mathcal{E}}[\![(F\ A)^{\sigma}]\!] \\
= & \quad \text{unfold} \\
& \_^{\vee\sigma} \circ (\mathrm{APPLY}\ \acute{\mathcal{E}}[\![F^{\tau\to\sigma}]\!]\ \acute{\mathcal{E}}[\![A^{\tau}]\!]) \\
= & \quad \text{abutting calculation} \\
& \mathrm{PUSH}\ \acute{\mathcal{E}}[\![A^{\tau}]\!]\ \grave{\mathcal{E}}[\![F^{\tau\to\sigma}]\!] \circ \check{\_} \\
= & \quad \text{extract} \\
& \grave{\mathcal{E}}[\![(F\ A)^{\sigma}]\!] \circ \check{\_}
\end{aligned}
\qquad\qquad
\begin{aligned}
& (\mathrm{APPLY}\ \acute{\mathcal{E}}[\![F^{\tau\to\sigma}]\!]\ \acute{\mathcal{E}}[\![A^{\tau}]\!]\ \eta)^{\vee\sigma}\ \xi \\
= & \quad \text{evaluate} \\
& (\acute{\mathcal{E}}[\![F^{\tau\to\sigma}]\!]\ \eta\ (\acute{\mathcal{E}}[\![A^{\tau}]\!]\ \eta)^{\vee\sigma}\ \xi \\
= & \quad \text{eureka} \\
& (\acute{\mathcal{E}}[\![F^{\tau\to\sigma}]\!]\ \eta\ (\acute{\mathcal{E}}[\![A^{\tau}]\!]\ \eta)^{\vee\tau\wedge\tau})^{\vee\sigma}\ \xi \\
= & \quad \text{fold} \\
& (\acute{\mathcal{E}}[\![F^{\tau\to\sigma}]\!]\ \eta)^{\vee\tau\to\sigma}\ ((\acute{\mathcal{E}}[\![A^{\tau}]\!]\ \eta)^{\vee\tau} \succ\!+ \xi) \\
= & \quad \text{IH twice} \\
& \grave{\mathcal{E}}[\![F^{\tau\to\sigma}]\!]\ \check{\eta}\ (\grave{\mathcal{E}}[\![A^{\tau}]\!]\ \check{\eta} \succ\!+ \xi) \\
= & \quad \text{synthesize} \\
& \mathrm{PUSH}\ \grave{\mathcal{E}}[\![A^{\tau}]\!]\ \grave{\mathcal{E}}[\![F^{\tau\to\sigma}]\!]\ \check{\eta}\ \xi
\end{aligned}$$

Thus by a constructive induction proof we have synthesized the following set of actions

$$\begin{array}{rcl}
\mathrm{ENTER}\ x\ \eta\ \xi & = & \eta\ x\ \xi \\
\mathrm{RETURN}\ c\ \eta\ [\,] & = & c \\
\mathrm{PUSH}\ a\ f\ \eta\ \xi & = & f\ \eta\ (a\ \eta \rightarrowtail\!\!+ \xi) \\
\mathrm{TAKE}\ x\ e\ \eta\ (a \rightarrowtail\!\!+ \xi) & = & e\ \eta[x := a]\ \xi
\end{array}$$

together with a new set of valuation functions.

$$\begin{array}{rcl}
\mathcal{E}[\![\_]\!] & \in & expr \to env \to func \\
\mathcal{E}[\![x]\!] & = & \mathrm{ENTER}\ x \\
\mathcal{E}[\![c]\!] & = & \mathrm{RETURN}\ c \\
\mathcal{E}[\![F\ A]\!] & = & \mathrm{PUSH}\ \mathcal{E}[\![A]\!]\ \mathcal{E}[\![F]\!] \\
\mathcal{E}[\![\lambda x.B]\!] & = & \mathrm{TAKE}\ x\ \mathcal{E}[\![B]\!] \\
\mathcal{M}[\![\_]\!] & \in & expr \to \mathbb{N} \\
\mathcal{M}[\![E^{\mathbb{N}}]\!] & = & \mathcal{E}[\![E]\!]\ \eta_0\ [\,]
\end{array}$$

The real connoisseur will spot the Krivine-machine [21, 58] in this description.

## 1.4.4 Environment trimming

In the above description environments are shared; $\mathrm{PUSH}\ a\ f\ \eta\ \xi = f\ \underline{\eta}\ (a\ \underline{\eta} \rightarrowtail\!\!+ \xi)$. An alternative is that each suspension has its own private environment consisting of precisely the values of its free variables.

Let $\mathcal{FV}\ E$ denote the set of free variables occurring in the expression $E$, and let $\eta \upharpoonright s$ be the restriction of environment $\eta$ to the variables appearing in the set $s$. Then a refinement for the above rule for function application may be expressed as

$$\mathrm{PUSH}\ \mathcal{E}[\![A]\!]\ \mathcal{E}[\![F]\!]\ \eta\ \xi \quad = \quad \mathcal{E}[\![F]\!]\ (\eta \upharpoonright \mathcal{FV}\ F)\ (\mathcal{E}[\![A]\!]\ (\eta \upharpoonright \mathcal{FV}\ A) \rightarrowtail\!\!+ \xi)$$

The restricted environments of argument expressions indicate the trimmed suspensions that are created for them, thus each suspension gets a tailor-made environment instead of sharing a single global one. For the function part, trimming merely shows to what extend variable slots in its environment may be reused.

If a function body contains no free occurrences of its binding variable, the rule for $\lambda$-expressions may be refined to

$$\begin{array}{ll}
 & \mathrm{TAKE}\ x\ \mathcal{E}[\![B]\!]\ \eta\ (a \rightarrowtail\!\!+ \xi) \\
= & \quad \text{evaluate} \\
 & \mathcal{E}[\![B]\!]\ (\eta[x := a] \upharpoonright \mathcal{FV}\ B)\ \xi \\
= & \quad x \notin \mathcal{FV}\ B \\
 & \mathcal{E}[\![B]\!]\ \eta\ \xi \\
= & \quad \text{synthesize} \\
 & \mathrm{DROP}\ x\ \mathcal{E}[\![B]\!]\ \eta\ (a \rightarrowtail\!\!+ \xi)
\end{array}$$

The net effect being that $a$ is removed from the stack.

### 1.4.5 An optimization for bound variables

When the argument of an application is a variable, a compile-time optimization is possible whether or not environments are trimmed or shared.

$$
\begin{array}{ll}
 & \mathcal{E}[\![\mathrm{F}\ \mathrm{x}]\!]\ \eta\ \xi \\
= & \quad \text{unfold} \\
 & \mathrm{PUSH}\ \mathcal{E}[\![\mathrm{x}]\!]\ \mathcal{E}[\![\mathrm{F}]\!]\ \eta\ \xi \\
= & \quad \text{unfold} \\
 & \mathcal{E}[\![\mathrm{F}]\!]\ \eta\ (\mathcal{E}[\![\mathrm{x}]\!]\ \eta \rightarrowtail \xi) \\
= & \quad \text{unfold and evaluate} \\
 & \mathcal{E}[\![\mathrm{F}]\!]\ \eta\ (\eta\ \mathrm{x} \rightarrowtail \xi) \\
= & \quad \text{synthesize} \\
 & \mathrm{PUSHARG}\ \mathrm{x}\ \mathcal{E}[\![\mathrm{F}]\!]\ \eta\ \xi
\end{array}
$$

As all other run-time actions make use of $\xi$, no other such optimization is possible.

### 1.4.6 Intermediate Code

As argued in the introduction, the semantic function $\mathcal{E}[\![\_]\!]$ can be partially evaluated into a compiler $\mathcal{C}[\![\_]\!] \in \mathit{expr} \rightarrow \mathrm{I\_code}$ and a lower level semantics $\mathcal{M}[\![\_]\!] \in \mathrm{I\_code} \rightarrow \mathit{env} \rightarrow \mathit{func}$ such that $\mathcal{E}[\![\_]\!] = \mathcal{M}[\![\_]\!] \circ \mathcal{C}[\![\_]\!]$. The advantage of using such intermediate code is that it leaves the choice between an implementation based on shared or private environments open, depending on $\mathcal{M}[\![\_]\!]$.

$$
\begin{array}{rcl}
\mathcal{C}[\![\_]\!] & \in & \mathit{expr} \rightarrow \mathrm{I\_code} \\
\mathcal{C}[\![\mathrm{x}]\!] & = & \mathrm{ENTER}\ \mathrm{x} \\
\mathcal{C}[\![\mathrm{c}]\!] & = & \mathrm{RETURN}\ \mathrm{c} \\
\mathcal{C}[\![\mathrm{F}\ \mathrm{A}]\!] & = & \mathrm{PUSH}\ \mathcal{A}[\![\mathrm{A}]\!];\mathcal{C}[\![\mathrm{f}]\!] \\
\mathcal{C}[\![\lambda \mathrm{x}.\mathrm{B}]\!] & = & \lambda \mathrm{x}.\{\mathcal{C}[\![\mathrm{B}]\!]\} \\
\mathcal{A}[\![\mathrm{x}]\!] & = & \mathrm{ARG}\ \mathrm{x} \\
\mathcal{A}[\![\mathrm{A}]\!] & = & \mathrm{SUSP}\ [\mathcal{FV}\ \mathrm{A}]\{\mathcal{C}[\![\mathrm{A}]\!]\}
\end{array}
$$

The expression $\mathit{twice} = \lambda \mathrm{f}.\lambda \mathrm{x}.\mathrm{f}\ (\mathrm{f}\ \mathrm{x})$ is compiled into

$$
\begin{array}{rcl}
\mathit{twice} & = & \lambda \mathrm{f}.\lambda \mathrm{x}.\{\mathrm{PUSH}\ (\mathrm{SUSP}\ \mathrm{fx});\mathrm{ENTER}\ \mathrm{f}\} \\
\mathrm{fx} & = & [\mathrm{f},\mathrm{x}]\{\mathrm{PUSH}\ (\mathrm{ARG}\ \mathrm{x});\mathrm{ENTER}\ \mathrm{f}\}
\end{array}
$$

## 1.5 Translating into deBruijn-form

In this section we will derive a compiler that translates simple $\lambda$-expressions into *de Bruijn* form [109]. For example

$$\begin{array}{rcl} I = \lambda x.x & \text{becomes} & \lambda.0 \\ K = \lambda x.\lambda y.x & \text{becomes} & \lambda.\lambda.1 \\ S = \lambda f.\lambda g.\lambda x.f\ x\ (g\ x) & \text{becomes} & \lambda.\lambda.\lambda.2\ 0\ (1\ 0) \end{array}$$

A de Bruijn-number $i$ indicates how long ago, in terms of binding actions, the identifier corresponding to $i$ was bound to its value. Thus in conventional terminology the de Bruijn-number of an identifier corresponds to its *nesting depth*, and operationally means that $i$ *static links* have to be traversed to fetch the value of the identifier.

The insight leading to an efficient implementation of environments is that (finite) mappings $env = var \rightarrow func$ can be represented by pairs of lists of identifiers and values, i.e. $var{*}\|func{*}$. The abstraction map $\oplus \in (var{*}\|func{*}) \rightarrow (var \rightarrow func)$ is defined informally as

$$[x_0 \dots x_{n-1}] \oplus [a_0 \dots a_{n-1}] \quad = \quad (\eta_0[x_{n-1} := a_{n-1}]) \dots [x_0 := a_0]$$

where $\eta_0$ is the empty environment. We use $xs$ as a meta-variable over $var{*}$ and $\eta$ as meta-variable over $func{*}$. Using $\oplus$, the definition of a function $Concr \in (env \rightarrow func) \rightarrow var{*} \rightarrow (func{*} \rightarrow func)$ leaves little choice

$$Concr\ m\ xs\ \eta \quad = \quad m\ (xs \oplus \eta)$$

We note in passing that this transformation is valid in an untyped context as well.

Using $Concr$ we derive a concrete version of the semantics $\grave{\mathcal{M}}[\![\_]\!]$ of expressions.

$$\begin{array}{ll} & \grave{\mathcal{M}}[\![E]\!] \\ = & \quad \text{demand} \\ & \acute{\mathcal{M}}[\![E]\!] \\ = & \quad \text{unfold} \\ & \acute{\mathcal{E}}[\![E]\!]\ \eta_0\ [\,] \\ = & \quad \eta_0 = [\,] \oplus [\,] \text{ then fold } Concr \\ & Concr\ \acute{\mathcal{E}}[\![E]\!]\ [\,]\ [\,]\ [\,] \\ = & \quad \text{assume theorem (1.2) holds} \\ & \grave{\mathcal{E}}[\![E]\!]\ [\,]\ [\,]\ [\,] \end{array}$$

Thus under the premise

$$Concr\ \acute{\mathcal{E}}[\![E]\!]\ xs \quad = \quad \grave{\mathcal{E}}[\![E]\!]\ xs \tag{1.2}$$

we have found a semantics based on deBruijn numbers. We will try to determine an $\grave{\mathcal{E}}[\![E]\!]$ that makes equality (1.2) hold by structural induction on $expr$.

The most inventive steps are for variables and abstraction

$$\begin{array}{ll}
 & \mathit{Concr}\ \mathcal{E}[\![x]\!]\ xs \\
= & \quad \text{unfold} \\
 & \mathit{Concr}\ (\mathrm{ENTER}\ x)\ xs \\
= & \quad \text{abutting calculation} \\
 & \mathrm{FETCH}\ (\mathit{position}\ x\ xs) \\
= & \quad \text{extract} \\
 & \grave{\mathcal{E}}[\![x]\!]\ xs
\end{array}$$

$$\begin{array}{ll}
 & \mathit{Concr}\ (\mathrm{ENTER}\ x)\ xs\ \eta\ \xi \\
= & \quad \text{unfold} \\
 & (xs \oplus \eta)\ x\ \xi \\
= & \quad \text{eureka} \\
 & \mathrm{FETCH}\ (\mathit{position}\ x\ xs)\ \eta\ \xi
\end{array}$$

where FETCH and $\mathit{position}$ are defined as

$$\begin{array}{rcl}
\mathrm{FETCH}\ 0\ (a \succ\!\!+ as) & = & as \\
\mathrm{FETCH}\ (i+1)\ (a \succ\!\!+ as) & = & \mathrm{FETCH}\ i\ as \\
\\
\mathit{position}\ x\ (x \succ\!\!+ xs) & = & 0 \\
\mathit{position}\ x\ (y \succ\!\!+ ys) & = & 1 + \mathit{position}\ x\ ys
\end{array}$$

Next we synthesize a concrete TAKE instruction.

$$\begin{array}{ll}
 & \mathit{Concr}\ \acute{\mathcal{E}}[\![\lambda x.E]\!]\ xs \\
= & \quad \text{unfold} \\
 & \mathit{Concr}\ (\mathrm{TA\acute{K}E}\ x\ \acute{\mathcal{E}}[\![E]\!])\ xs \\
= & \quad \text{abutting calculation} \\
 & \mathrm{TA\grave{K}E}\ \grave{\mathcal{E}}[\![E]\!]\ (x \succ\!\!+ xs) \\
= & \quad \text{extract} \\
 & \grave{\mathcal{E}}[\![\lambda x.E]\!]\ xs
\end{array}$$

$$\begin{array}{ll}
 & \mathit{Concr}\ (\mathrm{TA\acute{K}E}\ x\ \acute{\mathcal{E}}[\![E]\!])\ xs\ \eta\ (a \succ\!\!+ \xi) \\
= & \quad \text{unfold} \\
 & \mathrm{TA\acute{K}E}\ x\ \acute{\mathcal{E}}[\![E]\!]\ (xs \oplus \eta)\ (a \succ\!\!+ \xi) \\
= & \quad \text{evaluate} \\
 & \acute{\mathcal{E}}[\![E]\!]\ (xs \oplus \eta)[x := a]\ \xi \\
= & \quad \text{fold } \oplus \\
 & \acute{\mathcal{E}}[\![E]\!]\ (x \succ\!\!+ xs \oplus a \succ\!\!+ \eta)\ \xi \\
= & \quad \text{fold } \mathit{Concr} \\
 & \mathit{Concr}\ \acute{\mathcal{E}}[\![E]\!]\ (x \succ\!\!+ xs)\ (a \succ\!\!+ \eta)\ \xi \\
= & \quad \text{synthesize} \\
 & \mathrm{TA\grave{K}E}\ (\mathit{Concr}\ \acute{\mathcal{E}}[\![E]\!]\ (x \succ\!\!+ xs))\ \eta\ (a \succ\!\!+ \xi) \\
= & \quad \text{IH} \\
 & \mathrm{TA\grave{K}E}\ (\grave{\mathcal{E}}[\![E]\!]\ (x \succ\!\!+ xs))\ \eta\ (a \succ\!\!+ \xi)
\end{array}$$

The two remaining cases for constants and application are just tedious calculations and therefore omitted. Putting all pieces of the proof together yields the following semantics.

$$\begin{array}{rcl}
\mathcal{E}[\![x]\!]\ xs & = & \mathrm{FETCH}\ (\mathit{position}\ x\ xs) \\
\mathcal{E}[\![c]\!]\ xs & = & \mathrm{RETURN}\ c \\
\mathcal{E}[\![\lambda x.B]\!]\ xs & = & \mathrm{TAKE}\ \mathcal{E}[\![B]\!]\ (x \succ\!\!+ xs)
\end{array}$$

$$\mathcal{E}[\![\mathrm{F}\ \mathrm{A}]\!]\ xs \quad = \quad \mathrm{PUSH}\ (\mathcal{E}[\![\mathrm{A}]\!]\ xs)\ (\mathcal{E}[\![\mathrm{F}]\!]\ xs)$$

with actions

$$\begin{aligned}
\mathrm{FETCH}\ i\ [a_0 \ldots a_i \ldots]\ \xi &= a_i\ \xi \\
\mathrm{RETURN}\ c\ \eta\ \xi &= c \\
\mathrm{TAKE}\ e\ \eta\ (a \succ\!\!+ \xi) &= e\ (a \succ\!\!+ \eta)\ \xi \\
\mathrm{PUSH}\ a\ f\ \eta\ \xi &= f\ \eta\ (a\ \eta \succ\!\!+ \xi)
\end{aligned}$$

Our running example $twice$ will be translated into the code

$$\begin{aligned}
twice &= \mathrm{TAKE}\ (\mathrm{TAKE}\ (\mathrm{PUSH}\ \mathrm{fx}\ (\mathrm{ENTER}\ 1))) \\
\mathrm{fx} &= \mathrm{PUSH}\ x\ (\mathrm{ENTER}\ 1) \\
x &= \mathrm{ENTER}\ 0
\end{aligned}$$

## 1.6 Towards a concrete implementation

In order to construct a low-level implementation based on the deBruijn-form semantics, we must choose efficient realizations for the various components: code, environment, and stack. Encoding this in a functional meta-language has many drawbacks; functional languages are not suited to describe low-level activities. Update schemes as proposed by Meijer [72], are a high-level language especially designed for specifying low-level operations on pointers and sequences. One way of looking at update schemes is as a linear representation of graph rewrite rules. Furthermore it is relatively easy to transform update schemes into low-level languages such as C.

The implementation for $\mathcal{F}$ we envisage represents environments as linked lists LINK $a\ e$ of *suspensions*. A suspension SUSP $p\ e$ is a pair of a code address $p$ and an environment pointer $e$. The argument stack is realized as a real stack. The stack pointer SP points at the top of the argument stack. The current environment is represented by ENV while the programme counter PC points at the next instruction to execute.

As intended by the definition

$$\mathrm{TAKE}\ e\ \eta\ (a \succ\!\!+ \xi) \quad = \quad e\ (a \succ\!\!+ \eta)\ \xi$$

the TAKE-instruction pops the topmost argument from the stack and binds it in the environment. The update scheme for TAKE

$$\begin{array}{llllll}
\mathrm{PC}[\,p\,] & p[\,\mathrm{TAKE}\,]p' & \mathrm{ENV}[\,e\,] & \mathrm{SP}[\,s\,] & s[\,a\,]s' & \\
\Rightarrow & & & & & \\
\mathrm{PC}[\,p'\,] & & \mathrm{ENV}[\,e'\,] & \mathrm{SP}[\,s'\,] & & e'[\,\mathrm{LINK}\ a\ e\,]
\end{array}$$

makes all low-level details concerning pointers explicit that are invisible in the functional version. To see this we draw an edge between a location $l$ that appears inside a cell, such as $p$ in PC[ $p$ ], and the occurrence of $l$ as the address of a sequence of cells, like $p$ in $p$[ TAKE ]$p'$.

$$
\begin{array}{llllll}
\mathrm{PC}[\,.\,] & .[\,\mathrm{TAKE}\,]. & \mathrm{ENV}[\,e\,] & \mathrm{SP}[\,.\,] & .[\,a\,]. & \\
\Rightarrow & & & & & \\
\mathrm{PC}[\,.\,] & & \mathrm{ENV}[\,.\,] & \mathrm{SP}[\,.\,] & & .[\,\mathrm{LINK}\ a\ e\,]
\end{array}
$$

Similarly we can rephrase the other actions in terms of update schemes. The action ENTER $i$ is expanded into $i$ times DEREF followed by ENTER. A DEREF instruction dereferences the environment chain once.

$$
\begin{array}{llll}
\mathrm{PC}[\,p\,] & p[\,\mathrm{DEREF}\,]p' & \mathrm{ENV}[\,e\,] & e[\,\mathrm{LINK}\ a\ e'\,] \\
\Rightarrow & & & \\
\mathrm{PC}[\,p'\,] & & \mathrm{ENV}[\,e'\,] &
\end{array}
$$

When the right suspension is reached, it may be ENTER-ed. The current environment $e$ is replaced by the environment $e''$ found in the suspension $a$[ SUSP $p''$ $e''$ ] and execution continues at $p''$.

$$
\begin{array}{lllll}
\mathrm{PC}[\,p\,] & p[\,\mathrm{ENTER}\,]p' & \mathrm{ENV}[\,e\,] & e[\,\mathrm{LINK}\ a\ e'\,] & a[\,\mathrm{SUSP}\ p''\ e''\,] \\
\Rightarrow & & & & \\
\mathrm{PC}[\,p''\,] & & \mathrm{ENV}[\,e''\,] & &
\end{array}
$$

A pointer to a suspension built from code and the current environment is PUSH-ed onto the stack.

$$
\begin{array}{llllll}
\mathrm{PC}[\,p\,] & p[\,\mathrm{PUSH}\ p'\,]p'' & \mathrm{ENV}[\,e\,] & \mathrm{SP}[\,s\,] & & \\
\Rightarrow & & & & & \\
\mathrm{PC}[\,p''\,] & & & \mathrm{SP}[\,s'\,] & s'[\,a\,]s & a[\,\mathrm{SUSP}\ p'\ e\,]
\end{array}
$$

Finally, the RETURN instruction delivers its value in some result register and halts.

$$
\begin{array}{lll}
\mathrm{PC}[\,p\,] & p[\,\mathrm{RETURN}\ c\,] & \\
\Rightarrow & & \\
\mathrm{PC}[\,\mathrm{HALT}\,] & & \mathrm{RES}[\,c\,]
\end{array}
$$

The function $twice = \lambda f.\lambda x.f\ (f\ x)$ would be translated into the code

$$
\begin{array}{l}
twice[\,\mathrm{TAKE}, \mathrm{TAKE}, \mathrm{PUSH}\ fx, \mathrm{DEREF}, \mathrm{ENTER}\,] \\
fx[\,\mathrm{PUSH}\ x, \mathrm{DEREF}, \mathrm{ENTER}\,] \\
x[\,\mathrm{ENTER}\,]
\end{array}
$$

Osborne [81] has (manually) translated an update scheme specification for a similar abstract machine into C. The speed of the resulting implementation was of the same order of magnitude as that of Miranda[2].

[2]Miranda is a trademark of research software ltd.

# Chapter 2

# Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire

*Hey Mr.Tallyman*
*Tally me banana*
*Daylight come and me wanna go home*
George Clinton

Among the many styles and methodologies for the construction of computer programs, the Squiggol style in our opinion deserves attention from the functional programming community. The overall goal of Squiggol is to *calculate* programs from their specification in the way a mathematician calculates solutions to differential equations, or uses arithmetic to solve numerical problems.

It is not hard to state, prove and use laws for well-known operations such as addition, multiplication and —at the function level— composition. It is, however, quite hard to state, prove and use laws for arbitrarily recursively defined functions, mainly because it is difficult to refer to the recursion scheme in isolation. The algorithmic structure is obscured by using unstructured recursive definitions. We crack this problem by treating various recursion schemes as distinct higher order functions, giving each a notation of its own independent of the ingredients with which it constitutes a recursively defined function.

This philosophy is similar in spirit to the 'structured programming' methodology for imperative programming. The use of arbitrary goto's is abandoned in favor of structured control flow primitives such as conditionals and while-loops that replace fixed patterns of goto's, so that reasoning about programs becomes feasible and sometimes even elegant. For functional programs the question is which recursion schemes are to be chosen as a basis for a calculus of programs. We shall consider several recursion operators that are naturally associated with algebraic type definitions. A number of general theorems are proven about these operators and subsequently used to transform programs and prove their correctness.

Bird and Meertens [12, 67] have identified several laws for specific data types (most notably *finite* lists) using which they calculated solutions to various programming problems. By embedding the calculus into a categorical framework, Bird and Meertens' work on lists can be extended to arbitrary, inductively defined data types [63, 39]. Recently the group of Backhouse [6] has extended the calculus to a relational framework, thus covering indeterminacy.

Independently, Paterson [83, 84] has developed a calculus of functional programs similar in contents but very dissimilar in appearance (like many Australian animals) to the work referred to above. Actually if one pricks through the syntactic differences, the laws derived by Paterson are the same and in some cases slightly more general than those developed by the Squiggolers.

This chapter gives an extension of the theory to the context of lazy functional programming, i.e., for us a type is an $\omega$-cpo and we consider only continuous functions between types (categorically, we are working in the category CPO). Working in the category SET as done by for example Malcolm [63] or Hagino [48] means that finite data types (defined as initial algebras) and infinite data types (defined as final co-algebras) constitute two different worlds. In that case it is not possible to define functions by induction (catamorphisms) that are applicable to both finite and infinite data types, and arbitrary recursive definitions are not allowed. Working in CPO has the advantage that the carriers of initial algebras and final co-algebras coincide, thus there is a single data type that comprises both finite and infinite elements. The price to be paid however is that partiality of both functions and values becomes unavoidable. A cut back version this chapter has appeared as a joint paper with Fokkinga and Paterson in [70], a more formal treatment of the subject matter is given in [39] and in Fokkinga's thesis [37].

## 2.1 The data type of lists

We shall illustrate the recursion patterns of interest by means of the specific data type of cons-lists. So, the definitions given here are actually specific instances of those given in §2.5. Modern functional languages allow the definition of cons-lists over some type $A$ by putting:

$$A* \quad ::= \quad \mathrm{Nil} \mid \mathrm{Cons}\ (A \| A*)$$

In a conventional functional language the cartesian product $A\|A*$ would be written as $(A, A*)$. The recursive structure of this definition is employed when writing functions $\in A* \to B$ that destruct a list; these have been called *catamorphisms* (from the greek preposition $\kappa\alpha\tau\alpha$ meaning "downwards" as in "catastrophe"). *Anamorphisms* are functions $\in B \to A*$ (from the greek preposition $\alpha\nu\alpha$ meaning "upwards" as in "anabolism") that generate a list of type $A*$ from a seed from $B$. Functions of type $A \rightarrow B$ whose activation tree ([1], pp. 391) has the shape of a cons-list are called *hylomorphisms* (from the Aristotelian philosophy that form and matter are one, $\upsilon\lambda o\sigma$ meaning "dust" or "matter").

## Catamorphisms

Let $b \in B$ and $\oplus \in A||B \rightarrow B$, then the list-catamorphism $h \in A* \rightarrow B$ recursively replaces the constructor $Nil$ by the value $b$ and $Cons$ by the binary operator $\oplus$.

$$\begin{aligned} h\ \mathrm{Nil} &= b \\ h\ (Cons\ (a, as)) &= a \oplus (h\ as) \end{aligned} \tag{2.1}$$

In the notation of Bird&Wadler [13] one would write $h = foldr\ b\ (\oplus)$. We write catamorphisms by wrapping the relevant constituents between so called banana brackets:

$$h = (\!|b, \oplus|\!) \tag{2.2}$$

Countless list processing functions are readily recognizable as catamorphisms, for example $length \in A* \rightarrow \mathbb{N}$, or $filter\ p \in A* \rightarrow A*$, with $p \in A \rightarrow bool$.

$$\begin{aligned} length &= (\!|0, \oplus|\!)\ where\ a \oplus n = 1 + n \\ filter\ p &= (\!|\mathrm{Nil}, \oplus|\!) \\ & \quad where\ a \oplus as = Cons\ (a, as), \quad if\ p\ a \\ & \qquad\qquad\quad = as, \qquad\qquad\quad if\ \neg p\ a \end{aligned}$$

Separating the recursion pattern for catamorphisms $(\!|\_|\!)$ from its ingredients $b$ and $\oplus$ makes it feasible to reason about catamorphic programs in an algebraic way. For example the *Fusion Law* for catamorphisms over finite lists reads:

$$f \circ (\!|b, \oplus|\!) = (\!|c, \otimes|\!) \quad \Leftarrow \quad f\ b = c\ \wedge\ f\ (a \oplus as) = a \otimes (f\ as)$$

Without special notation pinpointing catas, such as $(\!|\_|\!)$ or $foldr$, we would be forced to formulate the fusion law as follows.

> Let $h, g$ be given by
>
> $$\begin{aligned} h\ \mathrm{Nil} &= b & g\ \mathrm{Nil} &= c \\ h\ (Cons\ (a, as)) &= a \oplus (h\ as) \qquad & g\ (Cons\ (a, as)) &= a \otimes (g\ as) \end{aligned}$$
>
> then $f \circ h = g$ if $f\ b = c$ and $f\ (a \oplus as) = a \otimes (f\ as)$.

A clumsy way of stating such a simple algebraic property.

## Anamorphisms

Given a predicate $p \in B \to \mathit{bool}$ and a function $g \in B \to A\|B$, the list-anamorphism $h \in B \to A*$ yields $\mathit{Nil}$ if $p\ b$ holds and otherwise constructs a list by $\mathit{Cons}$-ing the first argument $a$ of the pair $(a, b')$ generated by the seed $g\ b$ to the list returned by the recursive call $h\ b'$.

$$\begin{array}{rcll} h\ b & = & \mathit{Nil}, & \mathit{if}\ p\ b \\ & = & \mathit{Cons}\ (a, h\ b'), & \mathit{otherwise} \\ & & \mathit{where}\ (a, b') = g\ b & \end{array} \tag{2.3}$$

Anamorphisms are not well-known in the functional programming folklore, they are called $\mathit{unfold}$ by Bird&Wadler, who spend only few words on them. We denote anamorphisms by wrapping the relevant ingredients between concave lenses:

$$h = [\![ g, p ]\!] \tag{2.4}$$

Many important list-valued functions are anamorphisms; for example $\mathit{zip} \in A*\|B* \to (A\|B)*$ which 'zips' a pair of lists into a list of pairs.

$$\begin{array}{rcl} \mathit{zip} & = & [\![ g, p ]\!] \\ p\ (as, bs) & = & (as = \mathit{Nil}) \vee (bs = \mathit{Nil}) \\ g\ (\mathit{Cons}\ (a, as), \mathit{Cons}\ (b, bs)) & = & ((a, b), (as, bs)) \end{array}$$

Another anamorphism is $\mathit{iterate}\ f$ which given $a$, constructs the infinite list of iterated applications of $f$ to $a$.

$$\mathit{iterate}\ f = [\![ g, \mathit{false}^{\bullet} ]\!]\ \mathit{where}\ g\ a = (a, f\ a)$$

We use $c^{\bullet}$ to denote the constant function $\lambda x.c$.

Given $f \in A \to B$, the map function $f* \in A* \to B*$ applies $f$ to every element in a given list.

$$\begin{array}{rcl} f{*}\mathit{Nil} & = & \mathit{Nil} \\ f{*}(\mathit{Cons}\ (a, as)) & = & \mathit{Cons}\ (f\ a, f{*}as) \end{array}$$

Since a list appears at both sides of its type, we might suspect that map can be written both as a catamorphism and as an anamorphisms. Indeed this is the case. As catamorphism: $f* = (\!| \mathit{Nil}, \oplus |\!)$ where $a \oplus bs = \mathit{Cons}\ (f\ a, bs)$, and as anamorphism $f* = [\![ g, p ]\!]$ where $p\ as = (as = \mathit{Nil})$ and $g\ (\mathit{Cons}\ (a, as)) = (f\ a, as)$.

## Hylomorphisms

A recursive function $h \in A \to C$ whose call-tree is isomorphic to a cons-list, i.e., a linear recursive function, is called a hylomorphism. Let $c \in C$ and $\oplus \in B\|C \to C$ and

$g \in A \to B||A$ and $p \in A \to bool$ then these determine the hylomorphism $h$

$$\begin{aligned} h\ a &= c, && \text{if } p\ a \\ &= b \oplus (h\ a'), && otherwise \\ & \quad where\ (b, a') = g\ a \end{aligned} \tag{2.5}$$

This is exactly the same structure as an anamorphism except that $Nil$ has been replaced by $c$ and $Cons$ by $\oplus$. We write hylomorphisms by wrapping the relevant parts into envelopes.

$$h \;=\; [\![(c, \oplus), (g, p)]\!] \tag{2.6}$$

A hylomorphism corresponds to the composition of an anamorphism that builds the call-tree as an explicit data structure and a catamorphism that reduces this data object into the required value.

$$[\![(c, \oplus), (g, p)]\!] \;=\; (\!|c, \oplus|\!) \circ [\![g, p]\!]$$

A proof of this equality will be given in §2.5.4.

An archetypical hylomorphism is the factorial function:

$$\begin{aligned} fac &= [\![(1, \times), (g, p)]\!] \\ p\ n &= n = 0 \\ g\ (1+n) &= (1+n, n) \end{aligned}$$

## Paramorphisms

The hylomorphism definition of the factorial may be correct but is unsatisfactory from a theoretic point of view since it is not inductively defined on the data type $num ::= 0 \mid 1 + num$. There is however no 'simple' $\varphi$ such that $fac = (\!|\varphi|\!)$. The problem with the factorial is that it "eats its argument and keeps it too" [105], the brute force catamorphic solution would therefore have $fac'$ return a pair $(n, n!)$ to be able to compute $(n+1)!$.

*Paramorphisms* were investigated by Meertens [68] to cover this pattern of primitive recursion. For type $num$ a paramorphism is a function $h$ of the form:

$$\begin{aligned} h\ 0 &= b \\ h\ (1+n) &= n \oplus (h\ n) \end{aligned} \tag{2.7}$$

For lists a paramorphism is a function $h$ of the form:

$$\begin{aligned} h\ Nil &= b \\ h\ (Cons\ (a, as)) &= a \oplus (as, h\ as) \end{aligned}$$

We write paramorphisms by wrapping the relevant constituents in barbed wire $h = \{\!|b, \oplus|\!\}$, thus we may write $fac = \{\!|1, \oplus|\!\}$ where $n \oplus m = (1+n) \times m$. The function $tails \in A* \to A**$, which gives the list of all tail segments of a given list is defined by the paramorphism $tails = \{\!|Cons\ (Nil, Nil), \oplus|\!\}$ where $a \oplus (as, tls) = Cons\ (Cons\ (a, as), tls)$.

## 2.2 The category CPO

In this section we give a short overview of cpo's, continuous functions and least fixed points. The material is well known, but we could not resist including it because of the interesting similarities with reasoning on the program level.

### 2.2.1 Partially ordered sets

A partially ordered set or *poset* is a pair $(D, \sqsubseteq)$ consisting of a set $D$ together with a partial order $\sqsubseteq$ on $D$. A poset $X \subseteq D$ is a *chain* if all elements in $X$ are comparable, i.e. for all $x, x' \in X$ either $x \sqsubseteq x'$ or $x' \sqsubseteq x$. The least or *bottom* element of a poset, if any, is usually denoted by $\perp$.

Given $d, d' \in D$, their *join* $d \sqcup d'$ is the least element in $D$ that is greater than both $d$ and $d'$.

$$c = d \sqcup d' \quad \equiv \quad (\forall e :: c \sqsubseteq e \equiv d \sqsubseteq e \wedge d' \sqsubseteq e)$$

In general it is not true that every two elements in $D$ have joins. An easy to verify law concerning join is $\perp \sqcup d = d \sqcup \perp = d$. The *least upperbound* of a subset $X \subseteq D$ is denoted by $\sqcup/$, conventionally written as $\bigsqcup$.

$$c = \sqcup/X \quad \equiv \quad (\forall e :: c \sqsubseteq e \equiv (\forall x \in X :: x \sqsubseteq e))$$

Not every $X \subseteq D$ need have a lub.

### 2.2.2 CPO's

A poset $D$ is a complete partial order or *$\omega$-cpo* if it contains a bottom element and each chain in $D$ has a lub, so $\sqcup/X$ does exist for any chain $X \subseteq D$. The lub $\sqcup/$ of a chain $X$ satisfies the laws

$$\begin{aligned} \sqcup/\{\} &= \perp & (2.8) \\ \sqcup/(\{x\} \cup X) &= x \sqcup (\sqcup/X) & (2.9) \end{aligned}$$

(this is *not* a definition of $\sqcup/$ !)

A function $f \in D \to E$ is *monotonic* if it respects the ordering on $D$,

$$f\ d \sqsubseteq f\ d' \quad \Leftarrow \quad d \sqsubseteq d'$$

A function $f \in D \to E$ is *continuous* if it respects lubs of chains, let $X \subseteq D$ be a chain:

$$(f \circ \sqcup/)\ X \quad = \quad (\sqcup/ \circ f*)\ X$$

where $f{*}X = \{f\ x \mid x \in X\}$. The extension of a function $f \in D \to E$ to $f{*} \in D{*} \to E{*}$ satisfies the following equations:

$$\begin{aligned} f{*}\{\} &= \{\} \\ f{*}(\{d\} \cup D) &= \{f\ d\} \cup (f{*}D) \end{aligned}$$

Again, this is *not* a definition of $f{*}$. If $f$ is continuous then it follows that $f$ is monotonic by taking the chain $X = x \sqsubseteq x'$. An example of a monotonic function that is not continuous is:

$$\begin{aligned} f &\in (\mathbb{N} \cup \omega) \to \{0, 1\} \\ f\ \omega &= 1 \\ f\ n &= 0 \end{aligned}$$

This $f$ is clearly monotonic, but $(f \circ \sqcup/)\ \mathbb{N} = f\ \omega = 1$ while $(\sqcup/ \circ f{*})\ \mathbb{N} = 0$.

### 2.2.3 Least Fixed Points

An element $d \in D$ is a *fixed point* of $f \in D \to D$ if $f\ d = d$, it is a *least* fixed point if for any other fixed point $d'$ it holds that $d \sqsubseteq d'$.

Let $f \in D \to D$, then we define the set of iterated applications of $f$ as:

$$\mathit{iterate}\ f\ d = \{f^i\ d \mid i \in \mathbb{N}\}$$

The function $\mathit{iterate}\ f\ d$ satisfies the following law

$$\mathit{iterate}\ f\ d = \{d\} \cup (f{*} \circ \mathit{iterate}\ f)\ d \tag{2.10}$$

*Kleene's recursion theorem* states that any continuous function $f \in D \to D$ has as least fixed point

$$\mu f = (\sqcup/ \circ \mathit{iterate}\ f)\ \bot$$

First we show that $\mu f$ is indeed a fixed point of $f$

$$\begin{aligned} & f\ (\mu f) \\ = & \quad \text{unfold} \\ & (f \circ \sqcup/ \circ \mathit{iterate}\ f)\ \bot \\ = & \quad f \text{ is continuous, } \mathit{iterate}\ f\ \bot \text{ is a chain} \\ & (\sqcup/ \circ f{*} \circ \mathit{iterate}\ f)\ \bot \\ = & \quad d = \bot \sqcup d \\ & \bot \sqcup (\sqcup/ \circ f{*} \circ \mathit{iterate}\ f)\ \bot \\ = & \quad (2.9) \\ & \sqcup/\ (\{\bot\} \cup (f{*} \circ \mathit{iterate}\ f)\ \bot) \\ = & \quad (2.10) \end{aligned}$$

$$\begin{array}{ll} & (\sqcup/ \circ \mathit{iterate}\ f)\ \bot \\ = & \quad \text{fold} \\ & \mu f \end{array}$$

To show that $\mu f$ is also the *least* fixed point of $f$, we use the recursion induction rule of Park.

$$\mu f \sqsubseteq d \quad \Leftarrow \quad f\ d \sqsubseteq d \qquad \text{(RecInd)}$$

Assume that $d$ is also a fixed point of $f$ then $f\ d \sqsubseteq d$ and thus according to the above lemma $\mu f \sqsubseteq d$.

### 2.2.4 Calculating with fixed points

In the sequel we will never again refer to the explicit definition $\mu f = (\sqcup/ \circ \mathit{iterate}\ f)\ \bot$, instead we supply a handful of handy calculation rules for fixed points.

**Fixed point property** The vacuous fact that the least fixed point of $f$ is also a fixed point of $f$,

$$d = \mu f \quad \Rightarrow \quad d = f\ d \qquad \text{(FPProp)}$$

allows us to unwind recursively defined objects.

**Fixed point induction** The fixed point induction rule of Scott and de Bakker [23, 85], is a free theorem [106, 28, 93] of the fixed point operator $\mu \in (D \to D) \to D$.

Let $f \in A \to A$ be a continuous function and $P$ be a *chain complete* (or *inclusive* or *admissible* or *inductive*) predicate on $A$, then the following induction rule holds.

$$\frac{\begin{array}{l} \bullet\ P\ (\bot) \\ \bullet\ (\forall a \in A\ :\ P\ (a)\ :\ P\ (f\ a)) \end{array}}{P\ (\mu f)} \qquad (\mu\text{-ind})$$

In order to show that some property $P$ hold for recursively defined function $\mu f$, it suffices to show that the induction base $P(\bot)$ holds, and that assuming the induction hypothesis $P(a)$ the induction step to $P\ (f\ a)$ can be made. The $\mu$-induction rule is easily generalized to $n$-ary predicates.

A predicate is called *chain complete* if it respects lubs of chains, for any chain $X$:

$$(\wedge/ \circ P*)\ X \quad \Rightarrow \quad (P \circ \sqcup/)\ X$$

Clearly not every predicate is admissible, but if $f$ and $g$ are continuous the predicate $P\ (f, g) \equiv f = g$ is admissible and if $f$ is continuous and $g$ monotonic then $P\ (f, g) \equiv f \sqsubseteq g$ is admissible. The conjunction and disjunction of admissible predicates are admissible. Paulson [85] and Bird [10] (Chapter 7) provide more profound discussions on fixed point theory.

**Fixed point fusion** Specializing the fixed point induction rule to functions yields the fixed point fusion rule.

$$f\ (\mu g) = \mu h \quad \Leftarrow \quad f \text{ strict} \ \wedge\ f \circ g = h \circ f \qquad \text{(FPFusion)}$$

Theorem (FPFusion) appears under different names in many places[1] [96, 74, 27, 8, 49, 4, 16, 100, 47, 114].

**Rolling rule** The rolling rule is a direct consequence of (FPFusion).

$$f\ \mu(g \circ f) \quad = \quad \mu(f \circ g) \qquad \text{(Rolling)}$$

This role is part of the folklore.

**Total fusion** By analogy of 'loop fusion', the total fusion law allows two recursive functions to be combined into a single recursive function.

$$\mu f \circ \mu g = \mu h \quad \Leftarrow \quad f\ a \circ g\ b = h\ (a \circ b) \qquad \text{(TotalFusion)}$$

Rule (TotalFusion) can be proved by fixed point induction on $P(a, b, c) \equiv a \circ b = c$.

## 2.3 Functors

In the preceding section we have given specific notations for some recursion patterns in connection with the particular type of cons-lists. In order to define the notions of cata-, ana-, hylo- and paramorphism for arbitrary data types, we now present a generic theory of data types and functions on them. For this we consider a recursive data type (also called 'algebraic' data type in Miranda) to be defined as the least fixed point of a functor[2].

A bifunctor $\dagger$ is a binary operation taking types into types and functions into functions such that if $f \in A \to B$ and $g \in C \to D$ then $f \dagger g \in A \dagger C \to B \dagger D$, and which preserves identities and composition:

$$\begin{aligned} id \dagger id &= id \\ f \dagger g \circ h \dagger j &= (f \circ h) \dagger (g \circ j) \end{aligned}$$

Bifunctors are denoted by $\dagger, \ddagger, \S, \ldots$ A monofunctor is a unary type operation $\mathsf{F}$, which is also an operation on functions, $\mathsf{F} \in (A \to B) \to (A\mathsf{F} \to B\mathsf{F})$ that preserves the identity and composition. We use $\mathsf{F}, \mathsf{G}, \ldots$ to denote monofunctors. In view of the notation $A*$ we write the application of a functor as a postfix: $A\mathsf{F}$. In §2.6.1 we will show that $*$ is a functor indeed.

The data types found in all current functional languages can be defined by using the following basic functors.

[1] Other references are welcome.

[2] We give the definitions of various concepts of category theory only for the special case of the category CPO. Also 'functors' are really endo-functors, and so on.

**Product** The (lazy) product $D\|D'$ of two types $D$ and $D'$ and its operation $\|$ on functions are defined as:

$$\begin{aligned} D\|D' &= \{(d, d') \mid d \in D, d' \in D'\} \\ (f\|g)\ (d, d') &= (f\ d, g\ d') \end{aligned}$$

Closely related to the functor $\|$ are the left and right *projections* and the *split* combinator:

$$\begin{aligned} exl\ (d, d') &= d \\ exr\ (d, d') &= d' \\ (f \vartriangle g)\ d &= (f\ d, g\ d) \end{aligned}$$

The relation between the projections and split is best expressed by the equivalence

$$f = g \vartriangle h \quad \equiv \quad exl \circ f = g \ \wedge\ exr \circ f = h \qquad \text{(SplitCharn)}$$

Using $exl, exr$ and $\vartriangle$ we can define $f\|g$ as $f\|g = (f \circ exl) \vartriangle (g \circ exr)$. We can also define $\vartriangle$ using $\|$ and the doubling combinator $\Delta\ d = (d, d)$, since $f \vartriangle g = f\|g \circ \Delta$.

**Strictness** The strictifying functor $\_{\perp}$ adds an additional bottom element to a given domain $D$: $D_\perp = \{\perp\} \cup D$. Its associated operation on functions is the function:

$$\begin{aligned} f_\perp\ \perp &= \perp \\ f_\perp\ d &= f\ d, d \in D \end{aligned}$$

**Sum** The sum $D \mid D'$ of $D$ and $D'$ and the operation $\mid$ on functions are defined as:

$$\begin{aligned} D \mid D' &= (\{0\}\|D \cup \{1\}\|D')_\perp \\ (f \mid g)\ \perp &= \perp \\ (f \mid g)\ (0, d) &= (0, f\ d) \\ (f \mid g)\ (1, d') &= (1, g\ d') \end{aligned}$$

The arbitrarily chosen numbers 0 and 1 are used to 'tag' the values of the two summands so that they can be distinguished. Closely related to the functor $\mid$ are the *injections* and the *junc* combinator:

$$\begin{aligned} inl\ x &= (0, x) \\ inr\ y &= (1, y) \\ (f \triangledown g)\ \perp &= \perp \\ (f \triangledown g)\ (0, x) &= f\ x \\ (f \triangledown g)\ (1, y) &= g\ y \end{aligned}$$

with which we can write $f \mid g = (inl \circ f) \triangledown (inr \circ g)$. Using $\nabla$ which removes the tags from its argument, $\nabla\ \perp = \perp$ and $\nabla\ (i, x) = x$, we can define $f \triangledown g = \nabla \circ f \mid g$. The relationship between $inl, inr$ and $\triangledown$ is nicely expressed by the adjunction

$$f = g \triangledown h \quad \equiv \quad f \circ inl = g \ \wedge\ f \circ inr = h \ \wedge\ f \circ \perp = \perp \qquad \text{(SumCharn)}$$

**Arrow** The bifunctor $\to$ that forms the function space $D \to D'$ of continuous functions from $D$ to $D'$, has as operation on functions the 'wrapping' function:

$$(f \to g)\ h \quad = \quad g \circ h \circ f$$

Often we will use the alternative notation $(g \leftarrow f)\ h = g \circ h \circ f$, where we have swapped the arrow already so that upon application the arguments need not be moved, thus localizing the changes occurring during calculations. The functional

$$(f \overset{F}{\leftarrow} g)\ h \quad = \quad f \circ hF \circ g$$

wraps its F-ed argument between $f$ and $g$.

Closely related to the $\to$ are the *apply*, *curry* and *uncurry* combinators:

$$\begin{aligned} f^{\subset}\ x\ y &= f\ (x,y) \\ f^{\supset}\ (x,y) &= f\ x\ y \\ @\ (f,x) &= f\ x \\ &= id^{\supset}\ (f,x) \end{aligned}$$

Currying ($\_^{\subset}$) and uncurrying ($\_^{\supset}$) are each others inverses.

$$f = g^{\subset} \quad \equiv \quad f^{\supset} = g \qquad\qquad \text{(CuryUncurryIso)}$$

Note that $\to$ is contra-variant in its first argument, i.e. $(f \to g) \circ (h \to j) = (h \circ f) \to (g \circ j)$. This turns out be be very annoying and we will not allow $\to$ as a data type functor.

**Identity, Constants** The identity functor $I$ is defined on types as $DI = D$ and on functions as $fI = f$. Any type $D$ induces a functor with the same name $\underline{D}$, whose operation on objects is given by $C\underline{D} = D$, and on functions $f\underline{D} = id$.

**Lifting** For mono-functors $F, G$ and bi-functor $\dagger$ we define the mono-functors $FG$ and $F\dagger G$ by

$$\begin{aligned} x(FG) &= (xF)G \\ x(F\dagger G) &= (xF) \dagger (xG) \end{aligned}$$

for both types and functions $x$.

In view of the first equation we need not write parenthesis in $xFG$. Notice that in $(F\dagger G)$ the bi-functor $\dagger$ is 'lifted' to act on functors rather than on objects; $(F\dagger G)$ is itself a mono-functor.

**Sectioning** Analogous to the sectioning of binary operators, $(a\oplus)\ b = a \oplus b$ and $(\oplus b)\ a = a \oplus b$ we define sectioning of bi-functors $\dagger$;

$$\begin{aligned} (A\dagger) &= \underline{A}\dagger I \\ (f\dagger) &= f \dagger id \end{aligned}$$

hence $B(A\dagger) = A \dagger B$ and $f(A\dagger) = id \dagger f$. Similarly we can define sectioning of $\dagger$ in its second argument, i.e. $(\dagger B)$ and $(\dagger f)$.

It is not too difficult to verify the following two properties of sectioned functors:

$$\begin{aligned} (f\dagger) \circ g(A\dagger) &= g(B\dagger) \circ (f\dagger) \qquad \text{for all } f \in A \to B && (2.11) \\ (f\dagger) \circ (g\dagger) &= ((f \circ g)\dagger) && (2.12) \end{aligned}$$

Taking $f \dagger g = g \to f$, thus $(f\dagger) = (f\circ)$ gives some nice laws for function composition.

### 2.3.1 Laws for the basic combinators

There are various equations involving the above combinators, we state nothing but a few of these. In parsing an expression function composition has least binding power while $\|$ binds stronger than $|$.

$$\begin{array}{rcl@{\qquad}rcl}
exl \circ f\|g & = & f \circ exl & f \mid g \circ inl & = & inl \circ f \\
exl \circ f \vartriangle g & = & f & f \triangledown g \circ inl & = & f \\
exr \circ f\|g & = & g \circ exr & f \mid g \circ inr & = & inr \circ g \\
exr \circ f \vartriangle g & = & g & f \triangledown g \circ inr & = & g \\
(exl \circ h) \vartriangle (exr \circ h) & = & h & (h \circ inl) \triangledown (h \circ inr) & = & h \Leftarrow h \text{ strict} \\
exl \vartriangle exr & = & id & inl \triangledown inr & = & id \\
f\|g \circ h \vartriangle j & = & (f \circ h) \vartriangle (g \circ j) & f \triangledown g \circ h \mid j & = & (f \circ h) \triangledown (g \circ j) \\
f \vartriangle g \circ h & = & (f \circ h) \vartriangle (g \circ h) & f \text{ strict} \Rightarrow f \circ g \triangledown h & = & (f \circ g) \triangledown (f \circ h) \\
f\|g = h\|j & \equiv & f = h \wedge g = j & f \mid g = h \mid j & \equiv & f = h \wedge g = j \\
f \vartriangle g = h \vartriangle j & \equiv & f = h \wedge g = j & f \triangledown g = h \triangledown j & \equiv & f = h \wedge g = j
\end{array}$$

A nice law relating $\vartriangle$ and $\triangledown$ is the *abides law*:

$$(f \vartriangle g) \triangledown (h \vartriangle j) = (f \triangledown h) \vartriangle (g \triangledown j) \qquad (2.13)$$

### 2.3.2 Varia

The one element type is denoted **1** and can be used to model constants of type $A$ by nullary functions of type $\mathbf{1} \to A$. The only member of **1** called *void* is denoted by $()$.

In some examples we use for a given predicate $p \in A \to bool$, the function:

$$\begin{array}{rcll}
p? & \in & A \to A \mid A & \\
p?\ a & = & \bot, & p\ a = \bot \\
 & = & inl\ a, & p\ a = true \\
 & = & inr\ a, & p\ a = false
\end{array}$$

thus $f \triangledown g \circ p?$ models the familiar conditional **if** $p$ **then** $f$ **else** $g$ **fi**. The function VOID maps its argument to void: $\text{VOID}\ x = ()$. Some laws that hold for these functions are:

$$\begin{array}{rcl}
\text{VOID} \circ f & = & \text{VOID} \\
p? \circ x & = & x \mid x \circ (p \circ x)?
\end{array}$$

The following two simple, but handy λ-fusion laws will save a lot of work, since we don't have to invent variable-free versions of functions in order to calculate.

$$f \circ (\lambda x.E[x]) = \lambda x.f\ (E[x]) \qquad \text{(LambdaLeftFusion)}$$
$$(\lambda x.E[x]) \circ g = \lambda x.E[g\ x] \qquad \text{(LambdaRightFusion)}$$

We prove them both in one go.

$$\begin{array}{ll} & (f \circ \lambda x.E[x] \circ g)\ y \\ = & (f \circ \lambda x.E[x])\ (g\ y) \\ = & f\ (E[g\ y]) \\ = & (\lambda x.f\ E[g\ x])\ y \end{array}$$

Let F, G be functors and $\varphi_A \in AF \to AG$ for any type A. Such a φ is called a *polymorphic* function. A *natural transformation* is a family of functions $\varphi_A$ (omitting subscripts whenever possible) such that:

$$\forall f : f \in A \to B : \varphi_B \circ fF = fG \circ \varphi_A \qquad \text{(NatrTran)}$$

As a diagram the definition of a natural transformation becomes

$$\begin{tikzcd} AF \arrow[r, "\varphi_A"] \arrow[d, "fF"'] & AG \arrow[d, "fG"] \\ BF \arrow[r, "\varphi_B"'] & BG \end{tikzcd}$$

As a convenient shorthand for (NatrTran) we use $\varphi \in F \dot{\to} G$ to denote that φ is a natural transformation. The "Theorems For Free!" theorem of Wadler, deBruin and Reynolds [106, 28, 93] states that any function definable in the polymorphic λ-calculus is a natural transformation. If φ is defined using μ, one can only conclude that (NatrTran) holds for strict f.

## 2.4 Algebraic Data Types

After all this stuff on functors we have finally armed ourselves sufficiently to abstract from the peculiarities of cons-lists, and formalize recursively defined data types in general.

Let F be a monofunctor whose operation of functions is continuous, i.e., all monofunctors defined using the above basic functors or any of the map-functors introduced in § 2.6.1. Then there exists a type L and two strict functions $in_F \in LF \to L$ and $out_F \in L \to LF$ (omitting subscripts whenever possible) which are each others inverse such that

$$id = \mu(in \overset{F}{\leftarrow} out) \qquad \text{(Initial)}$$

[15, 95, 61, 99, 108, 39]. We let $\mu F$ denote the pair $(L, in)$ and say that it is "the least fixed point of F". Since $in$ and $out$ are each others inverses we have that $LF$ is isomorphic to $L$, and indeed $L$ is — upto isomorphism — a fixed point of $F$.

For example the functor $L$ defined on objects $X$ as

$$XL = \mathbf{1} \mid A \| X$$

defines the data type of cons-lists over $A$, for any type $A$; $(A*, in) = \mu L$ If we put $Nil = in \circ inl \in \mathbf{1} \to A*$ and $Cons = in \circ inr \in A \| A* \to A*$, we get the more familiar $(A*, Nil \triangledown Cons) = \mu L$.

Other examples of data types are binary trees with leaves of type $A$, that result from taking the least fixed point of functor

$$XT = \mathbf{1} \mid A \mid X \| X$$

Backward lists with elements of type $A$, or snoc-lists as they are sometimes called, are the least fixed point of

$$XL = \mathbf{1} \mid X \| A$$

i.e. $(A*, Nil \triangledown Snoc) = \mu L$. Natural numbers $(num, Zero \triangledown Succ)$ are specified as the least fixed point of $XN = \mathbf{1} \mid X$.

## 2.5 Recursion Schemes

Now that we have given a generic way of defining recursive data types, we can define cata-, ana-, hylo- and paramorphisms over arbitrary data types. Let $(L, in) = \mu F$, $\varphi \in AF \to A$, $\psi \in A \to AF$, $\xi \in (A \| L)F \to A$ then we define

$$
\begin{aligned}
(\!|\varphi|\!)_F &= \mu(\varphi \overset{F}{\leftarrow} out) && \text{(CataDef)}\\
[\!(\psi)\!]_F &= \mu(in \overset{F}{\leftarrow} \psi) && \text{(AnaDef)}\\
[\![\varphi, \psi]\!]_F &= \mu(\varphi \overset{F}{\leftarrow} \psi) && \text{(HyloDef)}\\
\{\!|\xi|\!\}_F &= \mu(\lambda f.\ \xi \circ (id \vartriangle f)F \circ out) && \text{(ParaDef)}
\end{aligned}
$$

When no confusion can arise we omit the F subscripts.

Definition (CataDef) agrees with the definition given in §2.1; where we wrote $(\!|e, \oplus|\!)$ we now write $(\!|e^\bullet \triangledown (\oplus)|\!)$.

Definition (AnaDef) agrees with the informal one given earlier on; the notation $[\!(g, p)\!]$ of §2.1 now becomes $[\!((VOID \mid g) \circ p?)\!]$.

Definition (HyloDef) agrees with the earlier one in the sense that taking $\varphi = c^\bullet \triangledown \oplus$ and $\psi = (VOID \mid g) \circ p?$ makes $[\![(c^\bullet, \oplus), (g, p)]\!]$ equal to $[\![\varphi, \psi]\!]$.

Definition (ParaDef) agrees with the description of paramorphisms as given in §2.1 in the sense that $\{\!|b, \oplus|\!\}$ equals $\{\!|b^\bullet \triangledown (\oplus)|\!\}$ here.

### 2.5.1 Program Calculation Laws

Rather than letting the programmer use explicit recursion, we encourage the use of the above fixed recursion patterns by providing a shopping list of laws that hold for these patterns. For each $\Omega$-morphism, with $\Omega \in \{\text{cata, ana, para}\}$, we give an *evaluation rule* which shows how such a morphism can be evaluated, and an *induction principle.* From the induction principle various specialized laws can be compiled; The *Uniqueness Property* is a canned induction proof for a given function to be a $\Omega$-morphism, the *fusion law* shows when the composition of some function with an $\Omega$-morphism is again an $\Omega$-morphism.

For hylomorphisms we prove that they can be split into an ana- and a catamorphism and show how computation may be shifted within a hylomorphism. A number of derived laws show the relation between certain cata- and anamorphisms. These laws are not valid in SET.

### 2.5.2 Catamorphisms

**Evaluation rule** The *evaluation rule* for catamorphisms follows from the fixed point property $x = \mu f \Rightarrow x = f\ x$:

$$(\!|\varphi|\!) \circ in \quad = \quad \varphi \circ (\!|\varphi|\!)F \qquad \text{(CataEval)}$$

It states how to evaluate an application of $(\!|\varphi|\!)$ to an arbitrary element of $L$ (returned by the constructor $in$); namely, apply $(\!|\varphi|\!)$ recursively to the argument of $in$ and then $\varphi$ to the result.

For cons-lists $(A*, Nil \triangledown Cons) = \mu L$ where $XL = \mathbf{1} \mid A \| X$ and $fL = id \mid id \| f$ with catamorphism $(\!|c \triangledown \oplus|\!)$ the evaluation rule reads:

$$\begin{aligned} (\!|c \triangledown \oplus|\!) \circ \bot &= \bot \\ (\!|c \triangledown \oplus|\!) \circ Nil &= c \\ (\!|c \triangledown \oplus|\!) \circ Cons &= \oplus \circ id \| (\!|c \triangledown \oplus|\!) \end{aligned}$$

i.e. the variable free formulation of (2.1). Notice that the constructors, here $Nil \triangledown Cons$ are used for parameter pattern matching.

**Induction principle for catamorphisms** The induction principle for catamorphism is given by

$$\frac{\begin{array}{l} \bullet\ f \circ \bot = g \circ \bot \\ \bullet\ (\forall x, y : f \circ x = g \circ y : f \circ \varphi \circ xF = g \circ \psi \circ yF) \end{array}}{f \circ (\!|\varphi|\!) = g \circ (\!|\psi|\!)} \qquad \text{(CataInd)}$$

It directly follows by fixed point induction on $P(x, y) \equiv f \circ x = g \circ y$.

**Fusion law** Explicit use of the induction principle should be avoided whenever possible, since the ritual steps, checking the premises and then declaring that by induction the theorem holds, hinder a smooth linear calculation. By partial evaluation (better partially performing) of induction proofs these steps are taken once and for all. The resulting theorem is a form of "canned induction". As an illustration we partially evaluate (CataInd) with $g := id$.

The first ritual step is to check the base case.

$$\begin{array}{ll} & f \circ \bot = g \circ \bot \\ \equiv & \quad g = id \\ & f \circ \bot = \bot \end{array}$$

The next ritual step is making the induction step, assuming the hypothesis $f \circ x = g \circ y \equiv f \circ x = y$.

$$\begin{array}{ll} & f \circ \varphi \circ xF = g \circ \psi \circ yF \\ \equiv & \quad g = id \\ & f \circ \varphi \circ xF = \psi \circ yF \\ \equiv & \quad \text{hypothesis} \\ & f \circ \varphi \circ xF = \psi \circ (f \circ x)F \\ \equiv & \quad \text{functor calculus} \\ & f \circ \varphi \circ xF = \psi \circ fF \circ xF \\ \equiv & \quad \text{assume } f \circ \varphi = \psi \circ fF \\ & true \end{array}$$

The third ritual step is the exclamation that by the induction principle for catamorphisms, the *fusion law* holds.

$$f \circ (\!|\varphi|\!) = (\!|\psi|\!) \quad \Leftarrow \quad f \circ \bot = \bot \ \wedge \ f \circ \varphi = \psi \circ fF \qquad \text{(CataFusion)}$$

**UP for catamorphisms** The *Uniqueness Property* can be used to prove the equality of two functions.

$$f = (\!|\xi|\!) \quad \equiv \quad f \circ \bot = (\!|\xi|\!) \circ \bot \ \wedge \ f \circ in = \xi \circ fF \qquad \text{(CataUP)}$$

The $\Rightarrow$ part of the proof for (CataUP) follows directly from the evaluation rule for catamorphisms. For the $\Leftarrow$ part we use the (CataInd) with $\varphi := in, \psi := in$ and $g := (\!|\xi|\!)$.

**Injective functions are catamorphisms** Let $f \in A \to B$ be a strict function with left-inverse $g$, then for any $\varphi \in AF \to A$ we can fuse $f$ with $(\!|\varphi|\!)$ since $f \circ \varphi = (f \circ \varphi \circ gF) \circ fF$.

$$f \circ (\!|\varphi|\!) = (\!|f \circ \varphi \circ gF|\!) \quad \Leftarrow \quad f \text{ strict} \ \wedge \ g \circ f = id \qquad (2.14)$$

Taking $\varphi = \mathit{in}$ we immediatly get that any strict injective function can be written as a catamorphism.

$$f = (\!|f \circ \mathit{in} \circ gF|\!)_F \quad \Leftarrow \quad f \text{ strict } \wedge\ g \circ f = \mathit{id} \tag{2.15}$$

Using this latter result we can write $\mathit{out}$ in terms of $\mathit{in}$ since $\mathit{out} = (\!|\mathit{out} \circ \mathit{in} \circ \mathit{in}L|\!) = (\!|\mathit{in}L|\!)$.

**Catamorphisms preserve strictness** The given laws for catamorphisms all demonstrate the importance of strictness, or generally of the behavior of a function with respect to $\perp$. The following "poor man's strictness analyser" for that reason can often be put into good use.

$$\mu F \circ \perp = \perp \quad \Leftarrow \quad \forall f : f \text{ strict} : F\ f \circ \perp = \perp \tag{2.16}$$

The proof of (2.16) is by fixed point induction over $P(F) \equiv F \circ \perp = \perp$.

Specifically for catamorphisms we have

$$(\!|\varphi|\!)_F \circ \perp = \perp \quad \equiv \quad \varphi \circ \perp = \perp$$

if F is strictness preserving. The $\Leftarrow$ part of the proof directly follows from (2.16) and the definition of catamorphisms. The other way around is shown as follows

$$\begin{array}{ll} & \perp \\ = & \quad \text{premise} \\ & (\!|\varphi|\!) \circ \perp \\ = & \quad \mathit{in} \circ \perp = \perp \\ & (\!|\varphi|\!) \circ \mathit{in} \circ \perp \\ = & \quad \text{(CataEval)} \\ & \varphi \circ (\!|\varphi|\!)F \circ \perp \\ = & \quad \text{F preserves strictness} \\ & \varphi \circ \perp \end{array}$$

## Examples

**Unfold-Fold** Many transformations usually accomplished by the unfold-simplify-fold technique can be restated using fusion. Let $(\mathbb{N}*, \mathit{Nil} \triangledown \mathit{Cons}) = \mu L$, where $XL = \mathbf{1} \mid \mathbb{N} \| X$ and $fL = \mathit{id} \mid \mathit{id} \| f$ be the type of lists of natural numbers. Using fusion we derive an efficient version of $\mathit{sum} \circ \mathit{squares}$ where $\mathit{sum} = (\!|0^\bullet \triangledown \mathit{add}|\!)$ adds together all elements of its argument list and $\mathit{squares} = (\!|\mathit{Nil} \triangledown (\mathit{Cons} \circ \mathit{sq} \| \mathit{id})|\!)$ squares each individual element of its argument list. Since $\mathit{sum}$ is strict we just start calculating aiming at the discovery of a $\psi$ that satisfies the condition of (CataFusion).

$$
\begin{array}{l}
\quad sum \circ \mathsf{Nil} \triangledown (\mathsf{Cons} \circ sq\|id) \\
= \\
\quad (sum \circ \mathsf{Nil}) \triangledown (sum \circ \mathsf{Cons} \circ sq\|id) \\
= \\
\quad 0^{\bullet} \triangledown (add \circ id\|sum \circ sq\|id) \\
= \\
\quad 0^{\bullet} \triangledown (add \circ (id \circ sq)\|(sum \circ id)) \\
= \\
\quad 0^{\bullet} \triangledown (add \circ (sq \circ id)\|(id \circ sum)) \\
= \\
\quad 0^{\bullet} \triangledown (add \circ sq\|id \circ id\|sum) \\
= \\
\quad (0^{\bullet} \circ id) \triangledown (add \circ sq\|id \circ id\|sum) \\
= \\
\quad 0^{\bullet} \triangledown (add \circ sq\|id) \circ id \mid id\|sum \\
= \\
\quad 0^{\bullet} \triangledown (add \circ sq\|id) \circ sumL
\end{array}
$$

and conclude that $sum \circ squares = \llparenthesis 0^{\bullet} \triangledown (add \circ sq\|id) \rrparenthesis$.

A slightly more complicated problem is to derive a one-pass solution for

$$average \quad = \quad div \circ sum \vartriangle length$$

Using the tupling lemma of Fokkinga [35]

$$\llparenthesis \varphi \rrparenthesis_L \vartriangle \llparenthesis \psi \rrparenthesis_L \quad = \quad \llparenthesis (\varphi \circ exlL) \vartriangle (\psi \circ exrL) \rrparenthesis \qquad \text{(BananaSplit)}$$

a simple calculation shows that $average = div \circ \llparenthesis (0^{\bullet} \triangledown add \circ id\|exl) \vartriangle (0^{\bullet} \triangledown succ \circ exr) \rrparenthesis$.

**Accumulating Arguments** An important item in the functional programmer's bag of tricks is the technique of *accumulating arguments*, where an extra parameter is added to a function to accumulate the result of the computation. Though stated here in terms of catamorphisms over cons-lists, the same technique is applicable to other data types and other kind of morphisms as well.

$$
\begin{array}{l}
\llparenthesis c^{\bullet} \triangledown \oplus \rrparenthesis\ l = \llparenthesis (c\otimes)^{\bullet} \triangledown \ominus \rrparenthesis\ l\ \nu_{\oplus}\ where\ (a \ominus f)\ b = f\ (a \odot b) \qquad (2.17) \\
\Leftarrow \\
a \otimes \nu_{\otimes} = a\ \wedge\ \bot \otimes a = \bot\ \wedge\ (a \oplus b) \otimes c = b \otimes (a \odot c)
\end{array}
$$

Theorem (2.17) follows from the fusion law by taking $Accu \circ \llparenthesis c^{\bullet} \triangledown \oplus \rrparenthesis = \llparenthesis (c\oplus)^{\bullet} \triangledown \ominus \rrparenthesis$ with $Accu\ a\ b = a \otimes b$.

Rewriting (2.17) into traditional notation makes the importance of the rule more clear, the transformed function is *tail-recursive*.

Let $\acute{h}$ be defined as

$$
\begin{array}{rcl}
\acute{h}\ \mathsf{Nil} & = & c \\
\acute{h}\ (\mathsf{Cons}(a, as)) & = & a \oplus (\acute{h}\ as)
\end{array}
$$

then $\acute{h}\ as = (\grave{h}\ as)\ \otimes\ \nu$ where

$$
\begin{array}{rcl}
\grave{h}\ \mathsf{Nil}\ b & = & c \oplus b \\
\grave{h}\ (\mathsf{Cons}(a, as))\ b & = & h\ as\ (a \odot b)
\end{array}
$$

A choice for $\otimes$ that works for any $\oplus$ is $a \otimes f = f\ a$ in which case $a \odot f = f \circ (a\oplus)$. This is known as *continuation passing style*, the continuation $f \circ (a\oplus)$ is an explicit functional representation of the 'return stack'.

Given the naive quadratic definition of $reverse \in A* \to A*$ as a catamorphism $(\!|Nil^{\bullet} \triangledown \oplus|\!)$ where $a \oplus as = as +\!\!+ (Cons\ (a, Nil))$, we can derive a linear time algorithm by instantiating (2.17) with $\oplus := +\!\!+$ and $\odot := Cons$ to get a function which accumulates the list being reversed as an additional argument: $(\!|id \triangledown \ominus|\!)$ where $(a \ominus as)\ bs = as\ (Cons\ (a, bs))$. Here $+\!\!+$ is the function that appends two lists, defined as $as +\!\!+ bs = (\!|id^{\bullet} \triangledown \oplus|\!)\ as\ bs$ where $a \oplus f\ bs = Cons\ (a, f\ bs)$.

In general catamorphisms of higher type $L \to (I \to S)$ form an interesting class by themselves as they correspond to *attribute grammars* [38].

### 2.5.3 Anamorphisms

**Evaluation rule** The evaluation rule for anamorphisms is given by:

$$out \circ [\![\psi]\!] \quad = \quad [\![\psi]\!]L \circ \psi \qquad \text{(AnaEval)}$$

It says what the result of an arbitrary application of $[\![\psi]\!]$ looks like: the constituents produced by applying $out$ can equivalently be obtained by first applying $\psi$ and then applying $[\![\psi]\!]L$ recursively to the result.

Anamorphisms are real old fusspots to explain. To instantiate (AnaEval) for cons-list we define the functions $hd \in A* \to A$ which delivers the first element of a nonempty list, the operation $tl \in A* \to A*$ which yields the tail of a nonempty list and the test $is\_nil$ which checks if a list is empty.

$$\begin{aligned} hd &= \bot \triangledown exl \circ out \\ tl &= \bot \triangledown exr \circ out \\ is\_nil &= true^{\bullet} \triangledown false^{\bullet} \circ out \end{aligned}$$

Assuming that $f = [\![VOID \mid (h \vartriangle t) \circ p?]\!]$ we find after a little calculation that:

$$\begin{aligned} is\_nil \circ f &= p \\ hd \circ f &= h \Leftarrow \neg p \\ tl \circ f &= t \Leftarrow \neg p \end{aligned}$$

which corresponds to the characterization of $unfold$ given by Bird and Wadler [13] on page 173.

**Induction Principle for anamorphisms** The induction rule for anamorphisms is slightly simpler then the one for catamorphisms since the base case need not be checked.

$$\frac{\bullet\ (\forall x, y\ :\ x \circ f = y \circ g\ :\ xF \circ \varphi \circ f = yF \circ \psi \circ g)}{[\![\varphi]\!] \circ f = [\![\psi]\!] \circ g} \qquad \text{(AnaInd)}$$

**Fusion law for anamorphisms** The strictness requirement that was needed for catamorphisms can be dropped in the anamorphism case. The dual condition of $f \circ \bot = \bot$ for strictness is $\bot \circ f = \bot$, which is vacuously true.

$$[\![\varphi]\!] \circ f = [\![\psi]\!] \quad \Leftarrow \quad \varphi \circ f = fF \circ \psi \qquad \text{(AnaFusion)}$$

This law can be proved from the induction principle by taking $g := id$.

**UP for anamorphisms** The UP for anamorphisms follows the evaluation rule and from (AnaFusion) by putting $\varphi := out$.

$$f = [\![\psi]\!] \quad \equiv \quad out \circ f = fF \circ \psi \qquad \text{(AnaUP)}$$

**Any surjective function is an anamorphism** The results (2.14) and (2.15) can be dualized for anamorphisms. Let $f \in B \to A$ a surjective function with right-inverse $g$, then for any $\psi \in A \to AL$ we have

$$[\![\psi]\!] \circ f = [\![gL \circ \psi \circ f]\!] \quad \Leftarrow \quad f \circ g = id \qquad (2.18)$$

since $\psi \circ f = fL \circ (gL \circ \psi \circ f)$. The special case where $\psi$ equals $out$ yields that any surjective function can be written as an anamorphism.

$$f = [\![gL \circ out \circ f]\!]_L \quad \Leftarrow \quad f \circ g = id \qquad (2.19)$$

As $in$ has right-inverse $out$, we can express $in$ using $out$ by $in = [\![outL \circ out \circ in]\!] = [\![outL]\!]$.

## Examples

Reformulated in the lenses notation, the function $iterate\ f$ becomes:

$$iterate\ f \quad = \quad [\![inr \circ id \vartriangle f]\!]$$

We have $[\![inr \circ id \vartriangle f]\!] = [\![VOID \mid id \vartriangle f \circ false^\bullet?]\!] (= [\![id \vartriangle f, false^\bullet]\!]$ in the notation of section 2.1).

Another useful list-processing function is $takewhile\ p$ which selects the longest initial segment of a list all whose elements satisfy $p$. In conventional notation:

$$\begin{array}{rcll} takewhile\ p\ Nil & = & Nil & \\ takewhile\ p\ (Cons\ a\ as) & = & Nil, & f\ \neg p\ a \\ & = & Cons\ a\ (takewhile\ p\ as), & otherwise \end{array}$$

The anamorphism definition may look a little daunting at first:

$$takewhile\ p \quad = \quad [\![inl \triangledown (VOID \mid id \circ (\neg p \circ exl)?) \circ out]\!]$$

The function $f\ while\ p$ contains all repeated applications of $f$ as long as predicate $p$ holds:

$$f\ while\ p \quad = \quad takewhile\ p \circ iterate\ f$$

Using the fusion law (after a rather long calculation) we can show that $f\ while\ p = [\![VOID \mid (id \vartriangle f) \circ \neg p?]\!]$.

### 2.5.4 Hylomorphisms

**Splitting Hylomorphisms** In order to prove that a hylomorphism can be split into an anamorphism followed by a catamorphism

$$[\![\varphi, \psi]\!] \quad = \quad (\!|\varphi|\!) \circ [\!(\psi)\!] \qquad \text{(HyloSplit)}$$

we can use the total fusion theorem (TotalFusion). The anamorphism $[\!(\psi)\!]$ builds an explicit F-shaped call-tree, which is subsequently reduced by catamorphism $(\!|\varphi|\!)$.

**Shifting law** Hylomorphisms are nice since their decomposability into a cata- and an anamorphism allows us to use the respective fusion laws to shift computation in or out of a hylomorphism. The following *shifting law* shows how computations can be shifted within a hylomorphism.

$$[\![\varphi \circ \xi, \psi]\!]_L = [\![\varphi, \xi \circ \psi]\!]_M \quad \Leftarrow \quad \xi \in L \dot{\to} M \qquad \text{(HyloShift)}$$

The proof of this theorem is straightforward.

$$\begin{array}{ll} & [\![\varphi \circ \xi, \psi]\!]_L \\ = & \quad \text{definition hylo} \\ & \mu(\lambda f.\varphi \circ \xi \circ fL \circ \psi) \\ = & \quad \xi \in L \dot{\to} M \\ & \mu(\lambda f.\varphi \circ fM \circ \xi \circ \psi) \\ = & \quad \text{definition hylo} \\ & [\![\varphi, \xi \circ \psi]\!]_M \end{array}$$

An admittedly humbug example of (HyloShift) shows how certain left linear recursive functions can be transformed into right linear recursive functions. Let $fL = id \mid f \| id$ and $fR = id \mid id \| f$ define the functors which express left respectively right linear recursion, then if $x \oplus y = y \oplus x$ we have

$$\begin{array}{ll} & [\![c \triangledown \oplus, f \mid (h \vartriangle t) \circ p?]\!]_L \\ = & [\![c \triangledown \oplus \circ SWAP, f \mid (h \vartriangle t) \circ p?]\!]_L \\ = & \quad SWAP \in L \dot{\to} R \\ & [\![c \triangledown \oplus, SWAP \circ f \mid (h \vartriangle t) \circ p?]\!]_R \\ = & [\![c \triangledown \oplus, f \mid (t \vartriangle h) \circ p?]\!]_R \end{array}$$

where $SWAP = id \mid (exr \vartriangle exl)$.

### 2.5.5 Relating cata- and anamorphisms

From the splitting and shifting law (HyloShift), (HyloSplit) and the fact that $(\!|\varphi|\!) = [\![\varphi, out]\!]$ and $[\!(\psi)\!] = [\![in, \psi]\!]$ we can derive a number of interesting laws which relate cata- and anamorphisms with each other [83].

$$(\!|\mathit{in}_M \circ \varphi|\!)_L = [\![\varphi \circ \mathit{out}_L]\!]_M \quad \Leftarrow \quad \varphi \in L \dot{\to} M \tag{2.20}$$

Using this law we can easily show that

$$\begin{aligned} (\!|\varphi \circ \psi|\!)_L &= (\!|\varphi|\!)_M \circ [\![\psi \circ \mathit{out}_L]\!]_M \Leftarrow \psi \in L \dot{\to} M & (2.21)\\ &= (\!|\varphi|\!)_M \circ (\!|\mathit{in}_M \circ \psi|\!)_L \Leftarrow \psi \in L \dot{\to} M & (2.22)\end{aligned}$$

$$\begin{aligned} [\![\varphi \circ \psi]\!]_M &= (\!|\mathit{in}_M \circ \varphi|\!)_L \circ [\![\psi]\!]_L \Leftarrow \varphi \in L \dot{\to} M & (2.23)\\ &= [\![\varphi \circ \mathit{out}_L]\!]_M \circ [\![\psi]\!]_L \Leftarrow \varphi \in L \dot{\to} M & (2.24)\end{aligned}$$

This set of laws will be used in §2.6.

From the total fusion theorem (TotalFusion) we can derive:

$$[\![\psi]\!]_F \circ (\!|\varphi|\!)_F = \mathit{id} \quad \Leftarrow \quad \psi \circ \varphi = \mathit{id} \tag{2.25}$$

Its dual companion ensures only partial correctness.

$$(\!|\varphi|\!)_F \circ [\![\psi]\!]_F \sqsubseteq \mathit{id} \quad \Leftarrow \quad \varphi \circ \psi = \mathit{id} \tag{2.26}$$

### Example: Reflecting binary trees

The type of binary trees with leaves of type $A$ is given by $(\mathit{tree}\ A, \mathit{in}) = \mu L$ where $XL = \mathbf{1} \mid A \mid X \| X$ and $fL = \mathit{id} \mid \mathit{id} \mid g \| g$. Reflecting a binary tree can be defined by: $\mathit{reflect} = (\!|\mathit{in} \circ SWAP|\!)$ where $SWAP = \mathit{id} \mid \mathit{id} \mid (\mathit{exr} \vartriangle \mathit{exl})$. A simple calculation proves that $\mathit{reflect} \circ \mathit{reflect} = \mathit{id}$.

$$\begin{aligned} & \mathit{reflect} \circ \mathit{reflect} \\ = & \quad SWAP \circ fL = fL \circ SWAP \\ & [\![SWAP \circ \mathit{out}]\!] \circ (\!|\mathit{in} \circ SWAP|\!) \\ = & \quad SWAP \circ \mathit{out} \circ \mathit{in} \circ SWAP = \mathit{id} \\ & \mathit{id} \end{aligned}$$

## 2.5.6 Paramorphisms

The *evaluation rule* for paramorphisms is

$$\{\!|\varphi|\!\} \circ \mathit{in} \quad = \quad \varphi \circ (\mathit{id} \vartriangle \{\!|\varphi|\!\})F \tag{ParaEval}$$

The *Induction Rule* for paramorphisms is given by

$$\frac{\begin{array}{l} \bullet\ f \circ \bot = g \circ \bot \\ \bullet\ (\forall x, y\ :\ f \circ x = g \circ y\ :\ f \circ \varphi \circ (\mathit{id} \vartriangle x)F = g \circ \psi \circ (\mathit{id} \vartriangle y)F) \end{array}}{f \circ \{\!|\varphi|\!\} = g \circ \{\!|\psi|\!\}} \tag{ParaInd}$$

The *UP* for paramorphisms is similar to that of catamorphisms:

$$f = \langle\!|\varphi|\!\rangle \quad \equiv \quad f \circ \bot = \langle\!|\varphi|\!\rangle \circ \bot \;\wedge\; f \circ in = \varphi \circ (id \vartriangle f)F \qquad \text{(ParaUP)}$$

The *fusion law* for paramorphisms reads

$$f \circ \langle\!|\varphi|\!\rangle = \langle\!|\psi|\!\rangle \quad \Leftarrow \quad f \text{ strict} \;\wedge\; f \circ \varphi = \psi \circ (id \| f)F \qquad \text{(ParaFusion)}$$

Any function $f \in L \to A$ (with $(L, in) = \mu F$) is a paramorphism.

$$f \quad = \quad \langle\!|f \circ in \circ exlF|\!\rangle$$

The usefulness of this theorem can be read from its proof.

$$\begin{array}{ll}
& \langle\!|f \circ in \circ exlF|\!\rangle \\
= & \quad \text{definition (ParaDef)} \\
& \mu(\lambda g. f \circ in \circ exlF \circ (id \vartriangle g)F \circ out) \\
= & \quad \text{functor calculus} \\
& \mu(\lambda g. f \circ in \circ out) \\
= & \\
& f
\end{array}$$

## Example: composing paramorphisms from ana- and catamorphisms

A nice result is that any paramorphism can be written as the composition of a cata- and an anamorphism. Let $(L, in) = \mu L$ be given, then define

$$\begin{array}{rcl}
XM & = & (L \| X)L \\
hM & = & (id \| h)L \\
(M, IN) & = & \mu M
\end{array}$$

For numbers defined as the least fixed point of functor $XL = \mathbf{1} \mid X$, we get $XM = (num \| X)L = \mathbf{1} \mid num \| X$, i.e. $(num*, in) = \mu M$, which is the type of lists of numbers.

Now define $preds \in L \to M$ as follows:

$$preds \quad = \quad [\![\Delta L \circ out_L]\!]_M$$

For numbers this yields $preds = [\![id \mid \Delta \circ out]\!]$, that is given a number $N = n$, the expression $preds\ N$ delivers the list $[n-1, \ldots, 0]$.

Using $preds$ we massage $(\!|\varphi|\!)_M \circ preds$ into paramorphical shape:

$$\begin{aligned}
&(\!|\varphi|\!)_M \circ \mathit{preds}\\
=\;&(\!|\varphi|\!)_M \circ [\![\Delta L \circ \mathit{out}_L]\!]_M\\
=\;&\mu(\lambda f.\varphi \circ fM \circ \Delta L \circ \mathit{out}_L)\\
=\;&\mu(\lambda f.\varphi \circ (\mathit{id}\|f)L \circ (\mathit{id} \vartriangle \mathit{id})L \circ \mathit{out}_L)\\
=\;&\mu(\lambda f.\varphi \circ (\mathit{id} \vartriangle f)L \circ \mathit{out}_L)\\
=\;&\langle\varphi\rangle_L
\end{aligned}$$

Thus $\langle\varphi\rangle_L = (\!|\varphi|\!)_M \circ \mathit{preds}$. Since $(\!|\mathit{IN}|\!)_M = \mathit{id}$ it follows that $\mathit{preds} = \langle\mathit{IN}\rangle_L$.

## 2.6 Parametrized Types

In §2.1 we have defined for $f \in A \to B$, the map function $f* \in A* \to B*$. Two laws for $*$ are $\mathit{id}* = \mathit{id}$ and $(f \circ g)* = f* \circ g*$. These two laws precisely state that $*$ is a functor. Another characteristic property of map is that it leaves the 'shape' of its argument unchanged. It turns out that any *parametrized* data type comes equipped with such a map functor. A parametrized type is a type defined as the least fixed point of a sectioned bifunctor. Contrary to Malcolms approach [63] map can be defined both as a catamorphism and as an anamorphism.

### 2.6.1 Maps

Let $\dagger$ be a bi-functor, then we define the functor $*$ (sometimes written as $\textcircled{\dagger}$) on objects $A$ as the parametrized type $A*$ *where* $(A*, \mathit{in}) = \mu(A\dagger)$, and on functions $f \in A \to B$ as:

$$\begin{aligned}
f* \;&=\; (\!|\mathit{in} \circ (f\dagger)|\!)_{(A\dagger)} &\quad (2.27)\\
&=\; (\!|\mathit{in} \circ f \dagger \mathit{id}|\!)_L \ \mathit{where}\ gL = \mathit{id} \dagger g
\end{aligned}$$

Expanding the banana-brackets shows that $f* = \mu(\lambda g.\mathit{in} \circ f \dagger g \circ \mathit{out})$ which immediately gives an alternative version of $f*$ as an anamorphism:

$$f* \;=\; [\![(f\dagger) \circ \mathit{out}]\!]_{(B\dagger)}$$

Functoriality of $f*$ is calculated as follows:

$$\begin{aligned}
&f* \circ g*\\
=\;&\quad \text{definition } *\\
&(\!|\mathit{in} \circ (f\dagger)|\!) \circ (\!|\mathit{in} \circ (g\dagger)|\!)\\
=\;&\quad (2.22)\\
&(\!|\mathit{in} \circ (f\dagger) \circ (g\dagger)|\!)\\
=\;&\quad (2.12)
\end{aligned}$$

$$
\begin{array}{ll}
 & (\!|\mathit{in} \circ ((f \circ g)\dagger)|\!) \\
= & \quad \text{definition } * \\
 & (f \circ g)*
\end{array}
$$

Maps are shape preserving. Define $\mathit{SHAPE} = \mathit{VOID}*$ then $\mathit{SHAPE} \circ f* = \mathit{VOID} \circ f* = \mathit{SHAPE}$.

For cons-list $(A*, \mathit{Nil} \triangledown \mathit{Cons}) = \mu(A\dagger)$ with $A \dagger X = \mathbf{1} \mid A \| X$ and $f \dagger g = \mathit{id} \mid f \| g$ we get $f* = [\![ f \dagger \mathit{id} \circ \mathit{out} ]\!]$. From (CataUP) we find that this conforms to the usual definition of map.

$$
\begin{array}{rcl}
f* \circ \bot & = & \bot \\
f* \circ \mathit{Nil} & = & \mathit{Nil} \\
f* \circ \mathit{Cons} & = & \mathit{Cons} \circ f \| f*
\end{array}
$$

Other important laws for maps are *factorization* [104] and *promotion* [12].

$$
\begin{array}{rcl}
(\!|\varphi|\!) \circ f* & = & (\!|\varphi \circ (f\dagger)|\!) \\
f* \circ [\![\psi]\!] & = & [\![(f\dagger) \circ \psi]\!]
\end{array}
\qquad
\begin{array}{r}
(2.28) \\
(2.29)
\end{array}
$$

$$
\begin{array}{rcl}
(\!|\varphi|\!) \circ f* & = & g \circ (\!|\chi|\!) \ \Leftarrow \ g \circ \chi = \varphi \circ f \dagger g \ \wedge \ g \text{ strict} \\
f* \circ [\![\psi]\!] & = & [\![\xi]\!] \circ g \ \Leftarrow \ \xi \circ g = f \dagger g \circ \psi
\end{array}
\qquad
\begin{array}{r}
(2.30) \\
(2.31)
\end{array}
$$

Now we know that $*$ is a functor, we can recognize that $\mathit{in} \in I\dagger* \stackrel{\cdot}{\to} *$ and $\mathit{out} \in * \stackrel{\cdot}{\to} I\dagger*$ are natural transformations.

$$
\begin{array}{rcl}
f* \circ \mathit{in} & = & \mathit{in} \circ f \dagger f* \\
\mathit{out} \circ f* & = & f \dagger f* \circ \mathit{out}
\end{array}
$$

## Iterate promotion

Recall the function $\mathit{iterate}\ f = [\![ \mathit{inr} \circ \mathit{id} \vartriangle f ]\!]$, the following law turns an $\mathcal{O}(n^2)$ algorithm into an $\mathcal{O}(n)$ algorithm, under the assumption that evaluating $g \circ f^n$ takes $n$ steps.

$$
g* \circ \mathit{iterate}\ f = \mathit{iterate}\ h \circ g \quad \Leftarrow \quad g \circ f = h \circ g \qquad (2.32)
$$

Law (2.32) is an immediate consequence of the promotion law for anamorphisms (2.31).

Interestingly we may also define $\mathit{iterate}$ as a cyclic list:

$$
\mathit{iterate}\ f\ x \quad = \quad \mu(\lambda xs.\mathit{Cons}\ (x, f{*}xs))
$$

and use fixed point fusion to prove (2.32).

### 2.6.2 Map-Reduce factorization

A data type $(A{*}, \mathit{in}) = \mu(A\dagger)$ with $A \dagger X = A \mid X\mathsf{F}$ is called a *free* $\mathsf{F}$-type over $A$. For a free type we can always write *strict* catas $(\!|\psi|\!)$ as $(\!|f \triangledown \varphi|\!)$ by taking $f = \psi \circ \mathit{inl}$ and $\varphi = \psi \circ \mathit{inr}$. For $f{*}$ we get

$$\begin{aligned} f{*} &= (\!|\mathit{in} \circ f \mid \mathit{id}|\!) \\ &= (\!|\mathit{tau} \triangledown \mathit{join} \circ f \mid \mathit{id}|\!) \\ &= (\!|\mathit{tau} \circ f \triangledown \mathit{join}|\!) \end{aligned}$$

where $\mathit{tau} = \mathit{in} \circ \mathit{inl}$ and $\mathit{join} = \mathit{in} \circ \mathit{inr}$.

If we define the *reduction* with $\varphi$ as

$$\varphi/ \quad = \quad (\!|\mathit{id} \triangledown \varphi|\!) \tag{2.33}$$

the factorization law (2.28) shows that catamorphisms on a free type can be factored into a map followed by a reduce. The reduction finishes what the map has left undone.

$$\begin{aligned} & (\!|f \triangledown \varphi|\!) \\ = \; & (\!|\mathit{id} \triangledown \varphi \circ f \mid \mathit{id}|\!) \\ = \; & (\!|\mathit{id} \triangledown \varphi|\!) \circ f{*} \\ = \; & \varphi/ \circ f{*} \end{aligned}$$

The fact that $\mathit{tau}$ and $\mathit{join}$ are natural transformations give evaluation rules for $f{*}$ and $\varphi/$ on free types.

$$\begin{array}{rclrcl} f{*} \circ \mathit{tau} &=& \mathit{tau} \circ f & \varphi/ \circ \mathit{tau} &=& \mathit{id} \\ f{*} \circ \mathit{join} &=& \mathit{join} \circ f{*}\mathsf{F} & \varphi/ \circ \mathit{join} &=& \varphi \circ (\varphi/)\mathsf{F} \end{array}$$

Early Squiggol was based completely on map-reduce factorization. Some of these laws from the good old days; *reduce promotion* and *map promotion*.

$$\begin{aligned} \varphi/ \circ \mathit{join}/ &= \varphi/ \circ (\varphi/){*} \\ f{*} \circ \mathit{join}/ &= \mathit{join}/ \circ f{*}{*} \end{aligned}$$

### 2.6.3 Monads

Any free type gives rise to a monad [63], in the above notation, $({*}, \mathit{tau} \in \mathsf{I} \dot{\to} {*}, \mathit{join}/ \in {*}{*} \dot{\to} {*})$ since:

$$\begin{aligned} \mathit{join}/ \circ \mathit{tau} &= \mathit{id} \\ \mathit{join}/ \circ \mathit{tau}{*} &= \mathit{id} \\ \mathit{join}/ \circ \mathit{join}/ &= \mathit{join}/ \circ \mathit{join}/{*} \end{aligned}$$

Wadler [107] gives a thorough discussion on the concepts of monads and their use in functional programming.

## 2.7 Mutual Recursion

Until now we have not considered mutual recursion, neither for functions nor for data types. The theory is easily extended to deal with mutual recursion. Mutual recursive *function* definitions like

$$\mathsf{f} = \mathsf{A}[\mathsf{f}, \mathsf{g}] \qquad \mathsf{g} = \mathsf{B}[\mathsf{f}, \mathsf{g}]$$

can be solved in two ways. The most straightforward techniques is by *simultaneous recursion* [64, 44].

$$(\mathsf{f}, \mathsf{g}) \quad = \quad \mu(\lambda(\mathsf{f}, \mathsf{g}).(\mathsf{A}[\mathsf{f}, \mathsf{g}], \mathsf{B}[\mathsf{f}, \mathsf{g}])) \tag{MutRecFunc}$$

The fixed points of $\mathsf{f}$ and $\mathsf{g}$ are constructed hand in hand[3]. An alternative method is to use *iterated recursion* [7, 24]

$$\begin{array}{rcl} \mathsf{f} & = & \mu(\lambda \mathsf{f}.\mathsf{A}[\mathsf{f}, \mu(\lambda \mathsf{g}.\mathsf{B}[\mathsf{f}, \mathsf{g}])]) \\ \mathsf{g} & = & \mu(\lambda \mathsf{g}.\mathsf{B}[\mathsf{f}, \mathsf{g}]) \end{array}$$

Naming $\mu(\lambda \mathsf{g}.\mathsf{B}[\mathsf{f}, \mathsf{g}])$ as $\mathsf{f}\ast$, we obtain an equivalent, but more suggestive formulation

$$\begin{array}{rcl} \mathsf{f} & = & \mu(\lambda \mathsf{f}.\mathsf{A}[\mathsf{f}, \mathsf{f}\ast]) \\ \mathsf{g} & = & \mathsf{f}\ast \end{array} \tag{ItRecFunc}$$

Note that $\ast$ is *not* a map-functor, though nearly. In this case the fixed point of $\mathsf{f}$ is constructed first, and subsequently used to find the solution of $\mathsf{g}$. The equivalence of (MutRecFunc) and (ItRecFunc) is more involved than expected. In the proof we let $\acute{\mathsf{f}}$ and $\acute{\mathsf{g}}$ denote $\mathsf{f}, \mathsf{g}$ as found by simultaneous recursion, and $\grave{\mathsf{f}}, \grave{\mathsf{g}}$ the $\mathsf{f}, \mathsf{g}$ found by iterated recursion. It is obvious that $\grave{\mathsf{f}} = \mathsf{A}[\grave{\mathsf{f}}, \grave{\mathsf{g}}]$ and $\grave{\mathsf{g}} = \mathsf{B}[\grave{\mathsf{f}}, \grave{\mathsf{g}}]$, i.e. $(\grave{\mathsf{f}}, \grave{\mathsf{g}})$ is a fixed point of $\lambda(\mathsf{f}, \mathsf{g}).(\mathsf{A}[\mathsf{f}, \mathsf{g}], \mathsf{B}[\mathsf{f}, \mathsf{g}])$. To show that $(\grave{\mathsf{f}}, \grave{\mathsf{g}})$ is also the least fixed point we reason

$$\begin{array}{ll}
 & \grave{\mathsf{f}} \sqsubseteq \acute{\mathsf{f}} \\
\equiv & \\
 & \mu(\lambda \grave{\mathsf{f}}.\mathsf{A}[\grave{\mathsf{f}}, \grave{\mathsf{f}}\ast]) \sqsubseteq \acute{\mathsf{f}} \\
\Leftarrow & \quad \text{(RecInd)} \\
 & \mathsf{A}[\acute{\mathsf{f}}, \acute{\mathsf{f}}\ast] \sqsubseteq \acute{\mathsf{f}} \\
\equiv & \quad \acute{\mathsf{f}} \text{ is a fixed point} \\
 & \mathsf{A}[\acute{\mathsf{f}}, \acute{\mathsf{f}}\ast] \sqsubseteq \mathsf{A}[\acute{\mathsf{f}}, \acute{\mathsf{g}}] \\
\Leftarrow & \quad \text{monotonicity of } \mathsf{A} \\
 & \acute{\mathsf{f}}\ast \sqsubseteq \acute{\mathsf{g}} \\
\equiv & \\
 & \mu(\lambda \grave{\mathsf{g}}.\mathsf{B}[\acute{\mathsf{f}}, \grave{\mathsf{g}}]) \sqsubseteq \acute{\mathsf{g}} \\
\Leftarrow & \quad \text{(RecInd)} \\
 & \mathsf{B}[\acute{\mathsf{f}}, \acute{\mathsf{g}}] \sqsubseteq \acute{\mathsf{g}} \\
\equiv & \quad \acute{\mathsf{g}} \text{ is a fixed point} \\
 & \mathsf{B}[\acute{\mathsf{f}}, \acute{\mathsf{g}}] \sqsubseteq \mathsf{B}[\acute{\mathsf{f}}, \acute{\mathsf{g}}] \\
\equiv & \\
 & \mathit{true}
\end{array}$$

[3]Technically speaking we are working in the product category $\mathsf{CPO}^2$.

Using the fact that $\grave{f} \sqsubseteq \acute{f}$ we can similarly show that $\grave{g} \sqsubseteq \acute{g}$.

By analogy of mutual recursive functions, mutual recursive *data types* like

$$A ::= in_{\dagger}\ A \dagger B \qquad B ::= in_{\ddagger}\ A \ddagger B$$

can be solved either simultaneously [64, 61] or iteratively [36, 113, 61]. Using simultaneous recursion gives

$$((A, in), (B, IN)) \quad = \quad \mu(\dagger \vartriangle \ddagger) \qquad \text{(MutRec)}$$

Catamorphisms on mutual recursive types are mutual recursive functions, for the above types $A$ and $B$ we get

$$(f, g) \quad = \quad \mu(\lambda(f, g).(\varphi \circ f \dagger g \circ out, \psi \circ f \ddagger g \circ OUT)) \qquad \text{(CataMut)}$$

which will be written as $f = (\!|\varphi|\!)_{\dagger}$ and $g = (\!|\psi|\!)_{\ddagger}$.

The *fusion law* for catamorphisms (CataMut) reads

$$\begin{array}{rcl} f \circ (\!|\varphi|\!)_{\dagger} = (\!|\chi|\!)_{\dagger} & & \\ \wedge\ g \circ (\!|\psi|\!)_{\ddagger} = (\!|\xi|\!)_{\ddagger} & \Leftarrow & f \circ \varphi = \chi \circ f \dagger g\ \wedge \\ & & g \circ \psi = \xi \circ g \ddagger f \end{array} \qquad \text{(FusionMutRec)}$$

The iterative solution of mutual recursive data types will be derived by finding appropriate functors such that the iterative version of the catamorphisms (CataMut) (interpreted as recursive functions) can be defined as catamorphisms on the types resulting from taking the separate least fixed points of the derived functors. When rewriting the mutual recursive catamorphisms as defined in (CataMut) into iterative form, we get

$$\begin{array}{rcl} f & = & \mu(\lambda f.\varphi \circ f \dagger f\ast \circ out) \\ g & = & f\ast \end{array}$$

where $f\ast = \mu(\lambda g.\psi \circ f \ddagger g \circ OUT)$. Provided that $B = A\textcircled{\scriptsize\ddagger}$, the map factorization law (2.28) says that $f\ast$ equals

$$f\ast \quad = \quad (\!|\psi|\!)_{(A\ddagger)} \circ f\textcircled{\scriptsize\ddagger}$$

where $\textcircled{\scriptsize\ddagger}$ is the map functor induced by $\ddagger$:

$$\begin{array}{rcl} f\textcircled{\scriptsize\ddagger} & = & (\!|IN \circ f \ddagger id|\!)_{(A\ddagger)} \\ (B = A\textcircled{\scriptsize\ddagger}, IN) & = & \mu(A\ddagger) \end{array}$$

For $f$ we calculate

$$\begin{array}{ll} & f \\ = & \quad \text{unfold } f \\ & \mu(\lambda f.\varphi \circ f \dagger f\ast \circ out) \\ = & \quad \text{unfold } f\ast \end{array}$$

$$
\begin{array}{ll}
 & \mu(\lambda f.\varphi \circ f \dagger (\llparenthesis\psi\rrparenthesis_{(A\ddagger)} \circ f\textcircled{\ddagger}) \circ \mathrm{out}) \\
= & \qquad \dagger \text{ functor} \\
 & \mu(\lambda f.\varphi \circ \mathrm{id} \dagger \llparenthesis\psi\rrparenthesis_{(A\ddagger)} \circ f \dagger f\textcircled{\ddagger} \circ \mathrm{out}) \\
= & \qquad \text{fold, assuming that } (A, \mathrm{in}) = \mu(\mathsf{I}\dagger\textcircled{\ddagger}) \\
 & \llparenthesis\varphi \circ \mathrm{id} \dagger \llparenthesis\psi\rrparenthesis_{(A\ddagger)}\rrparenthesis_{\mathsf{I}\dagger\textcircled{\ddagger}}
\end{array}
$$

Hence we have derived that the *iterative* solution of mutual recursive types is given by

$$
\begin{array}{rcl}
(A, \mathrm{in}) & = & \mu(\mathsf{I}\dagger\textcircled{\ddagger}) \\
(B = A\textcircled{\ddagger}, \mathrm{IN}) & = & \mu(A\ddagger)
\end{array}
$$

## 2.8 The Induction Principle for Algebraic Data types

Fixed point induction is useful when we want to prove properties of recursively defined functions. However, not everything we would like can (easily) be proved using fixed point induction, for example $\mu(\mathrm{in} \overset{F}{\leftarrow} \mathrm{out}) = \mathrm{id}$. Remember that we have cunningly used this property in proofs for laws like *Uniqueness* to make the induction go.

If we want to prove a certain property for all $x \in L$, it is often more convenient to use *structural induction* [89] then to use fixed point induction.

Let $(L, \mathrm{in}) = \mu F$ and $P \subseteq L$ an admissible predicate, then the structural induction principle (StructInd) follows by fixed point induction on the predicate $Q(f) \equiv \forall x \in L :: P(f\ x)$ with $f = \mathrm{in} \overset{F}{\leftarrow} \mathrm{out}$ [85].

$$
\frac{\begin{array}{l}\bullet\ P(\bot) \\ \bullet\ \forall x \in LF : PF(x) : P(\mathrm{in}\ x)\end{array}}{\forall x \in L :: P(x)} \qquad \text{(StructInd)}
$$

It is illustrative to look at a concrete case of (StructInd), for example cons-lists $(A*, \mathrm{Nil} \triangledown \mathrm{Cons}) = \mu L$ with $XL = \mathbf{1} \mid A||X$. Instantiating the second premise gives

$$
\begin{array}{ll}
 & \forall x \in (A*)L : PL(x) : P(\mathrm{Nil} \triangledown \mathrm{Cons}\ x) \\
\equiv & \forall x \in \mathbf{1} \mid A||A* : x \in \mathbf{1} \mid A||P : P(\mathrm{Nil} \triangledown \mathrm{Cons}\ x) \\
\equiv & (\forall x \in \mathbf{1} : x \in \mathbf{1} : P(\mathrm{Nil})) \land (\forall (a, x) \in A||A* : a \in A \land x \in P : P(\mathrm{Cons}\ (a, x))) \\
\equiv & P(\mathrm{Nil}) \ \land \ (\forall a \in A, x \in A* : P(x) : P(\mathrm{Cons}\ (a, x)))
\end{array}
$$

Thus the structural induction principle for cons-lists reads

$$
\frac{\begin{array}{l}\bullet\ P(\bot) \\ \bullet\ P(\mathrm{Nil}) \\ \bullet\ \forall a \in A, x \in A* : P(x) : P(\mathrm{Cons}\ (a, x))\end{array}}{\forall x \in A* :: P(x)}
$$

## 2.9 Higher order functions wreck calculations

In this section we will pause a while to explain how higher order functions play havoc with our desire to eliminate explicit induction proofs in favor of nice calculational developments.

### 2.9.1 Banana split

Remember the banana split theorem

$$(\!|\varphi|\!)_F \vartriangle (\!|\psi|\!)_F \quad = \quad (\!|(\varphi \circ exl F) \vartriangle (\psi \circ exr F)|\!)$$

This works well for catamorphisms of type $L \rightarrow A$, however suppose we have $(\!|\varphi|\!) \in L \rightarrow (A \rightarrow B)$ and $(\!|\psi|\!) \in L \rightarrow (C \rightarrow D)$, then the tupled catamorphism $(\!|(\varphi \circ exl L) \vartriangle (\psi \circ exr L)|\!)$ has type $L \rightarrow (A \rightarrow B)\|(C \rightarrow D)$. What we would like to have is an operation $\mathbin{\underline{\blacktriangle}}$ such that $(\!|\varphi|\!)\mathbin{\underline{\blacktriangle}}(\!|\psi|\!) \in L \rightarrow (A\|C) \rightarrow (B\|D)$. Type considerations suggest to take

$$(\!|\varphi|\!)\mathbin{\underline{\blacktriangle}}(\!|\psi|\!) \quad = \quad (\|) \circ (\!|\varphi|\!) \vartriangle (\!|\psi|\!) \qquad \text{(AGF)}$$

By further massaging of the suggested definition for $\mathbin{\underline{\blacktriangle}}$ we get

$$
\begin{array}{ll}
 & (\|) \circ (\!|\varphi|\!) \vartriangle (\!|\psi|\!) \\
= & \quad \text{(BananaSplit)} \\
 & (\|) \circ (\!|(\varphi \circ exl L) \vartriangle (\psi \circ exr L)|\!) \\
= & \quad \text{assume } (\|) \text{ has left inverse} \\
 & (\!|(\|) \circ (\varphi \circ exl F) \vartriangle (\psi \circ exr F) \circ \|^{-1} F|\!) \\
= & \quad \text{simplify} \\
 & (\!|(\|) \circ \varphi\|\psi \circ (exl \leftarrow id \vartriangle \bot)F \vartriangle (exr \leftarrow \bot \vartriangle id)F|\!)
\end{array}
$$

Not a very sexy definition for such a simple operation (see also §4.2.4). The left-inverse $\|^{-1}$ of the product combinator $(\|)$ is synthesized by reasoning as follows.

$$
\begin{array}{ll}
 & \|^{-1}\ (f\|g) \\
= & \quad \text{wish} \\
 & (f, g) \\
= & \quad \text{introduce } \| \\
 & (exl \circ f\|g \circ id \vartriangle \bot, exr \circ f\|g \circ \bot \vartriangle id) \\
= & \quad \text{definition } \leftarrow \text{ and } \vartriangle \\
 & ((exl \leftarrow id \vartriangle \bot) \vartriangle (exr \leftarrow \bot \vartriangle id))\ (f, g)
\end{array}
$$

Hence $\|^{-1} = (exl \leftarrow id \vartriangle \bot) \vartriangle (exr \leftarrow \bot \vartriangle id)$ will do.

### 2.9.2 Compiler Calculation

In subsequent chapters many calculations have the following format. Given $\mathit{Abs} \in A \to B$ and $\mathit{Conc} \in B \to A$ such that $\mathit{Abs} \circ \mathit{Conc} = \mathrm{id}$, we want to derive $\mathit{Abs} \circ (\!|\psi|\!)$ from $\mathit{Abs} \circ \mathit{Conc} \circ (\!|\varphi|\!)$. This can be done in essentially two different ways.

$$\begin{array}{ll}
 & \mathit{Abs} \circ \mathit{Conc} \circ (\!|\varphi|\!) \\
= & \qquad \mathit{Conc} \circ \varphi = \psi \circ \mathit{Conc}\mathsf{F} \\
 & \mathit{Abs} \circ (\!|\psi|\!)
\end{array}
\qquad\qquad
\begin{array}{ll}
 & \mathit{Abs} \circ \mathit{Conc} \circ (\!|\varphi|\!) \\
= & \qquad \mathit{Abs} \circ \mathit{Conc} = \mathrm{id} \\
 & \mathit{Abs} \circ (\!|\mathit{Conc} \circ \varphi \circ \mathit{Abs}\mathsf{F}|\!)
\end{array}$$

The leftmost alternative usually give the desired solution for $\psi$. More precisely, the underlying action algebra is unsatisfactory.

Now let $(\!|\varphi|\!) \in L \to (B \to B)$, and try to derive $\mathit{Conc} \to \mathit{Abs} \circ (\!|\psi|\!)$ from $\mathit{Conc} \to \mathit{Abs} \circ \mathit{Abs} \to \mathit{Conc} \circ (\!|\varphi|\!)$. There are three ways to continue, of which we only show two.

$$\begin{array}{ll}
 & \mathit{Conc} \to \mathit{Abs} \circ \mathit{Abs} \to \mathit{Conc} \circ (\!|\varphi|\!) \\
= & \qquad \mathit{Abs} \to \mathit{Conc} \circ \varphi = \psi \circ (\mathit{Abs} \to \mathit{Conc})\mathsf{F} \\
 & \mathit{Conc} \to \mathit{Abs} \circ (\!|\psi|\!)
\end{array}$$

This one normally does not yield a satisfying solution for $\psi$, to find one we must work a little harder.

$$\begin{array}{ll}
 & (\mathit{Conc} \to \mathit{Abs} \circ \mathit{Abs} \to \mathit{Conc} \circ (\!|\varphi|\!))l \\
= & \\
 & \mathit{Abs} \circ \mathit{Conc} \circ (\!|\varphi|\!)\ l \circ \mathit{Abs} \circ \mathit{Conc} \\
= & \\
 & \mathit{Abs} \circ \mathit{Conc} \circ (\!|\varphi|\!)\ l \\
= & \qquad \mathit{Conc} \circ (\!|\varphi|\!)\ l = (\!|\psi|\!)\ l \circ \mathit{Conc} \\
 & \mathit{Abs} \circ (\!|\psi|\!)\ l \circ \mathit{Conc} \\
= & \\
 & (\mathit{Conc} \to \mathit{Abs} \circ (\!|\psi|\!))\ l
\end{array}$$

The question remains how to find an elegant and efficient proof for the step

$$\mathit{Conc} \circ (\!|\varphi|\!)\ l \quad = \quad (\!|\psi|\!)\ l \circ \mathit{Conc}.$$

As a first attempt we try to prove using (CataInd) that $\mathrm{id} \to a \circ (\!|\varphi|\!) = b \to \mathrm{id} \circ (\!|\psi|\!)$. The induction base requires that $a$ is strict. No progress can be made in the induction step, so that the following theorem remains.

$$\begin{array}{lll}
\mathrm{id} \to a \circ (\!|\varphi|\!) = b \to \mathrm{id} \circ (\!|\psi|\!) & \Leftarrow & a \circ \bot = \bot\ \wedge \\
 & & (\forall f, g\ :\ \mathrm{id} \to a \circ f = b \to \mathrm{id} \circ g\ : \\
 & & \mathrm{id} \to a \circ \varphi \circ f\mathsf{F} = b \to \mathrm{id} \circ \psi \circ g\mathsf{F})
\end{array}$$

Hence we are bound to conclude that we can not save any work over an explicit induction proof. Trying to apply "Theorems for free" does not help either. Straightforward expansion of Wadler's rules [106] leads to

$$\frac{\bullet\ (\forall x, y :: (x, y) \in (f \to g)\mathsf{F} \Rightarrow g \circ \varphi\ x = \psi\ y \circ f)}{\forall l \in \mu\mathsf{F} :: g \circ (\!|\varphi|\!)\ l = (\!|\psi|\!)\ l \circ f}$$

without knowing the functor F it seems impossible to simplify the expression $(x, y) \in (f \to g)\mathsf{F}$.

## 2.10 Continuous Algebras

Technically speaking our notion of data type comes comes very close to the concept of initial continuous algebra as introduced by the ADJ group [43] and Reynolds [92]. A *continuous* F-*algebra* is a pair $(D, \varphi)$ where the *carrier* $D$ is a cpo and the *operation* $\varphi$ is a strict function of type $D\mathsf{F} \to D$. A *homomorphism* $h$ from $(D, \varphi)$ to $(E, \psi)$ is a strict function such that $h \circ \varphi = \psi \circ h\mathsf{F}$. An algebra $(L, in)$ is *initial* if for any algebra $(D, \varphi)$ there is a unique homomorphism $h$ between $(L, in)$ and $(D, \varphi)$. From (CataUnique) it follows that $h$ equals $(\!|\varphi|\!)$ and that $\mu\mathsf{F}$ yields the initial F-algebra.

Dualizing the above reasoning gives that an F-*co-algebra* is a pair $(D, \varphi)$ with $\varphi$ a not necessarily strict function of type $D \to D\mathsf{F}$, that a co-homomorphism $h$ between $(D, \varphi)$ and $(E, \psi)$ is a function such that $\varphi \circ h = h\mathsf{F} \circ \psi$ and that $(L, out)$ is the final co-algebra with $[\![\varphi]\!]$ the unique co-homo to any $(D, \varphi)$.

The nice thing about working in CPO is that the carriers of the initial F-algebra and the final F-co-algebra coincide. The fly in the ointment is that we cannot calculate with properties induced from initiality as this would mean that we have to re*strict* ourselves too much.

## 2.11 Conclusions

We have considered various patterns of recursive definitions, and have presented a lot of laws that hold for the functions so defined. Although we have illustrated the laws and the recursion operators with examples, the usefulness for practical program calculation might not be evident to every reader. Hopefully the rest of this thesis will be convincing.

There are more aspects to program calculation than just a series of combining forms (like $(\!|\_|\!), [\![\_]\!], \{\!\!\{\_\}\!\!\}, [\![\_, \_]\!]$) and laws about them. For calculating large programs one certainly needs high level algorithmic theorems. The work reported here provides the necessary tools to develop such theorems. For the theory of lists Bird [11] has started to do so, and with success.

Another aspect of program calculation is machine assistance. Our experience —including that of our colleagues— shows that the size of formal manipulations is much greater than in most textbooks of mathematics; it may well be comparable in size to "computer algebra" as done in systems like MACSYMA, Maple, Mathematica etc. Fortunately, it also appears that most manipulations are easily automated and, moreover, that quite a few equalities depend on natural transformations. Thus in several cases type checking alone suffices. Clearly machine assistance is fruitful and does not seem to be too difficult.

Finally we observe that category theory has provided several notions and concepts that were indispensable to get a clean and smooth theory; for example, the notions of functor and natural transformation. Without doubt there is much more categorical knowledge that can be useful for program calculation; we are just at the beginning of an exciting development.

**Acknowledgements** Ross Paterson, Maarten Fokkinga and Graham Hutton provided many useful remarks on draft versions of this chapter.

Chapter 3

# From Pr/T-nets to Update Schemes

*It's been a long time commin*
*But I know a change is gonna come*
Otis Redding

This chapter introduces and relates two computational models; Predicate/Transition nets and update schemes. General (Petri) net theory [86] is a theory of structure and behavior of dynamic systems that stresses causality and distributedness of states and changes. Update schemes [72, 81] were proposed as a high-level formalism for describing low-level operations in abstract machines.

Meijer [72] used update schemes to describe the implementation of *Programmar*, a compiler-compiler based on *Extended Affix Grammars* [112]. Osborne [81, 80] has continued the work in the area of using update schemes as a formalism for specifying abstract machines. Ultimately he wants to produce a compiler for (a restricted class) of update schemes. We take a slightly different route by considering update schemes as a general formalism for specifying (concurrent) computations. In this thesis, update schemes will be used only to specify abstract machines. The use of update schemes as a formalism for specifying concurrent processes will be left as a topic for future research.

We will first discuss the Predicate/Transition net philosophy of describing dynamic systems. Then we define update schemes by taking different, yet equally plausible, assumptions than those underlying net theory.

## 3.1 Pr/T nets

The Predicate/Transition net (*Pr/T net*) [41] computational model describes a computation by means of a set of dynamically evolving predicates. Before plunging into theory first some examples.

### 3.1.1 Informal introduction

Nowadays divorce is quite common. The occurrence of the divorce of $\mathit{John}$ and $\mathit{Mary}$ changes the relations

$$\mathit{Married}\{\ldots,(\mathit{John},\mathit{Mary}),\ldots\} \qquad \mathit{Single}\{\ldots\}$$

to the modified relations

$$\mathit{Married}\{\ldots,\ldots\} \qquad \mathit{Single}\{\mathit{John},\mathit{Mary},\ldots\}$$

Pr/T nets are schemes for expressing such changes in a structured and formal manner.

Usually variable relations are represented by circles called *places* which are *marked* with those *tokens* for which the relation currently holds. Places can be grouped together to express a rule of change, called a *transition*. The graphical representation of a transition is a box connected to its input and output places by means of directed arrows. The box may be decorated with a guarding expression. We propose a textual representation for Pr/T-nets. A transition that expresses divorce for example is

$$\mathit{Divorce}: \ \mathit{Married}\{(m,f)\} \quad \Rightarrow \quad \mathit{Single}\{m,f\}$$

A Pr/T net is formed by connecting a number of places by means of transitions and providing an initial distribution of tokens over the net. A possible distribution of tokens over a Pr/T net is called a *case*.

The transition $\mathit{Divorce}$ can be embedded into a simple model of reincarnating life described by the following set of transitions:

$$\begin{array}{rlcll}
\mathit{Birth}: & \mathit{Dead}\{x\} & \Rightarrow & \mathit{Single}\{x\} & \\
\mathit{Die}_s: & \mathit{Single}\{x\} & \Rightarrow & \mathit{Dead}\{x\} & \\
\mathit{Marry}: & \mathit{Single}\{x,y\} & \Rightarrow & \mathit{Married}\{(x,y)\} & \\
\mathit{Divorce}: & \mathit{Married}\{(x,y)\} & \Rightarrow & \mathit{Single}\{x,y\} & \\
\mathit{Die}_m: & \mathit{Married}\{(x,y)\} & \Rightarrow & \mathit{Single}\{x\} & \mathit{Dead}\{y\} \\
\mathit{Die}'_m: & \mathit{Married}\{(x,y)\} & \Rightarrow & \mathit{Single}\{y\} & \mathit{Dead}\{x\}
\end{array}$$

Some properties that can be derived formally by merely looking at the form of the above net are

- the system is live, at any instance there is a transition that can occur.
- the total number of souls is invariant.

Another example is semaphores. Let $\mathrm{BS}\{p\}$ denote that process $p$ wants to enter the critical section, the condition $\mathrm{CS}\{p\}$ indicates that $p$ is occupying the critical section while $\mathrm{AS}\{p\}$ denotes that $p$ just left the critical section. Then the following net ensures that the critical section contains at most $n$ tokens where $n$ is the initial marking of place $S \subseteq \mathbb{N}$.

$$\begin{array}{llllll} P: & S\{v+1\} & BS\{p\} & \Rrightarrow & S\{v\} & CS\{p\} \\ V: & S\{v\} & CS\{p\} & \Rrightarrow & S\{v+1\} & AS\{p\} \end{array}$$

Transition P may occur if the value of semaphore S is at least 1, and some process $p$ wants to enter the critical section. Then in one atomic step, the value of the semaphore is decremented and $p$ enters the critical section. Transition V takes $p$ out of the critical section while increasing the value of the semaphore.

### 3.1.2 Formal definitions

Now that we have seen a few informal examples of Pr/T nets, the time has come to turn to a more formal treatment.

**syntax** The syntax of Pr/T nets is given by the grammar

$$\begin{array}{rcl} \mathrm{Pr/T\ net} & ::= & \mathrm{transition}{*}\ \mathrm{case} \\ \mathrm{transition} & ::= & \mathrm{name}:\ \mathrm{place}{*}\ \{\mathrm{guard}\} \Rrightarrow\ \mathrm{place}{*} \\ \mathrm{place} & ::= & \mathrm{predicate}\{\mathrm{token}{*}\} \\ \mathrm{token} & ::= & (\mathrm{term}, \ldots, \mathrm{term}) \end{array}$$

A *term* is an expression built from constants, function symbols and variables. A *ground* term is a term without variables. A *ground token* is a token without variables, i.e. a tuple of ground terms. A *guard* is a boolean term, tautological guards may be omited. A *case* is a set of ground places, i.e., a mapping from predicate symbols to sets of ground tokens. A *substitution* is a mapping from variables to terms.

**concession, redex** A transition may occur, or has concession if its guarding expression yields true, all of its pre-conditions and none of its post-conditions hold. Let $u = n : \ \mathrm{pre}\ \{\mathrm{guard}\} \Rrightarrow \ \mathrm{post}$ be a consistently renamed, 'fresh', copy of a transition of a given net, $\varphi$ a substitution and C a case. The pair $\Delta = (u, \varphi)$ is called a *redex*, or $u$ has *concession* at C under $\varphi$, iff

- $\varphi\ \mathrm{guard}$ holds,
- ${}^{\bullet}\Delta \subseteq C$; a token may only leave a relation of which it is a member,
- $\Delta^{\bullet} \cap C = \emptyset$; a token may only enter a relation of which it is not yet a member.

where ${}^{\bullet}\Delta = \varphi\ \mathrm{pre}$ and $\Delta^{\bullet} = \varphi\ \mathrm{post}$ are respectively the *pre-* and the *postconditions* of $\Delta$.

**Firing** a redex $\Delta = (u, \varphi)$ changes the current case into its successor by removing the preconditions ${}^\bullet\Delta$ from C and adding the postconditions $\Delta^\bullet$ in one atomic step:

$$C[u\rangle_\varphi C' \quad \equiv \quad C' = (C \setminus {}^\bullet\Delta) \cup \Delta^\bullet$$

We say that transition $u$ has *occurred*, or that $u$ has *fired*.

### 3.1.3 Relations between redexes

The next subsections discuss various relationships that may hold between redexes.

**ordered** Two redexes $\Delta$ and $\Gamma$ are *ordered*, notation $\Delta; \Gamma$, if $\Delta^\bullet \cap {}^\bullet\Gamma \neq \emptyset$. Ordered redexes cannot have concession at the same case and $\Delta$ must precede $\Gamma$ in any firing sequence. A simple example of ordered redexes is

$$\begin{array}{lcl} \mathrm{Ping}\{\} & \Rightarrow & \mathrm{Pong}\{\} \\ \mathrm{Pong}\{\} & \Rightarrow & \mathrm{Ping}\{\} \end{array}$$

starting (arbitrarily) at case $\mathrm{Ping}$. Note that $\mathrm{Ping}$ and $\mathrm{Pong}$ are nullary predicates (booleans) that either hold or don't hold.

**conflict** Two redexes are in *conflict* if they have concession at the same time but firing one takes concession from the other. More formally conflict between $\Delta$ and $\Gamma$ is expressed by: ${}^\bullet\Delta \cap {}^\bullet\Gamma \neq \emptyset$ (forward conflict) or $\Delta^\bullet \cap \Gamma^\bullet \neq \emptyset$ (backward conflict). Marriage is full of conflict, there may be several candidates for a marriage, but choosing a partner makes all other choices impossible.

**concurrent, no order** Two redexes are *independent* if they have no common pre- and post-conditions

$$({}^\bullet\Delta \cup \Delta^\bullet) \cap ({}^\bullet\Gamma \cup \Gamma^\bullet) = \emptyset$$

Independent redexes don't interfere with each other and hence can be fired *concurrently*. Let $\Delta = \{\Delta_i\}$ be a set of pairwise independent redexes, with $U = \{u \mid (u, \varphi) \in \Delta\}$, $\Phi = \{\varphi \mid (u, \varphi) \in \Delta\}$, ${}^\bullet\Delta = \cup/\{{}^\bullet\Delta_i \mid \Delta_i \in \Delta\}$ and $\Delta^\bullet = \cup/\{\Delta_i^\bullet \mid \Delta_i \in \Delta\}$, then

$$C[U\rangle_\Phi C' \quad \equiv \quad C' = (C \setminus {}^\bullet\Delta) \cup \Delta^\bullet$$

Independent redexes may be fired in any order. Let $C, C'$ be such that $C[V\rangle C'$ and $V \subseteq U$, then there exists a $C''$ such that $C[V\rangle C''$ and $C''[U \setminus V\rangle C'$.

## 3.2 Update Schemes

We now discuss some of the assumptions underlying Pr/T nets, propose alternative ones and examine consequences of replacing the old premises by the new.

### 3.2.1 Cells instead of Predicates

In a Pr/T net the state of a system is modeled by means of dynamically evolving predicates. Often we want to model state as a set of *cells* which may hold only a single value, i.e., a conventional digital computer's store. Moreover cells may be allocated or dismissed in the course of a computation. In a Pr/T net the total number of relations is invariant. A *configuration* in an update scheme computation is a (partial) function from a set of *locations* to a set of cells which may hold a single value. The total number of occupied locations may grow or shrink due to a transition.

In order to talk about *sequences* of cells we assume that the set of locations comes equipped with two functions $\mathrm{pred}$ and $\mathrm{succ}$ both of type $\mathrm{loc} \to \mathrm{loc}$ giving the successor and the predecessor of a given location respectively. The functions $\mathrm{pred}$ and $\mathrm{succ}$ satisfy $(\mathrm{pred} \circ \mathrm{succ})\ p = p$ whenever $\mathrm{succ}\ p$ is defined (and dually $(\mathrm{succ} \circ \mathrm{pred})\ p = p$ if $\mathrm{pred}\ p$ is defined). A sequence of cells $t$ between two locations $p$ and $q$ in some configuration C, notation $p[\,t\,]q \in C$, is defined as the list $[C\ p, (C \circ \mathrm{succ})\ p, \ldots, (C \circ \mathrm{pred})\ q]$. When no confusion can arise either of the boundary locations of a *locator* $p[\,t\,]q$ may be omitted. The expression $\mathrm{succ}^n\ p$ may be abbreviated as $p + n$ while $p - n$ may be used as a shorthand for $\mathrm{pred}^n\ p$.

### 3.2.2 Non-destructive reading and tests

The second difference between Pr/T nets and update schemes lies in the notions of concession and firing. A Pr/T net transition has concession only if all its pre-conditions and none of its post-conditions hold. Then when the transition occurs, the pre-conditions cease to hold while the post-conditions start to hold. The interpretation of a configuration as a computer store strongly suggests that the occurrence of a transition does not invalidate all of the pre-conditions; reading the value of a cell does not make it loose its contents. Rather the effect of a transition is the *minimal* amount of change to the configuration that makes its post-conditions hold.

The requirement that pre- and post-conditions of a Pr/T net transition are disjoint excludes side-conditions or tests. In an update scheme overlapping pre- and post-conditions will be allowed

### 3.2.3 Formal introduction

**syntax** The syntax of update schemes is given the grammar

$$\begin{array}{rcl} \mathrm{script} & ::= & \mathrm{scheme}{*}\ \mathrm{configuration} \\ \mathrm{scheme} & ::= & \mathrm{name} :\ \mathrm{locator}{*}\ \{\mathrm{guard}\} \Rightarrow\ \mathrm{locator}{*} \\ \mathrm{locator} & ::= & \mathrm{term}[\,\mathrm{term}\,]\mathrm{term} \end{array}$$

A *configuration* is a (partial) function $\in \mathit{loc} \to \mathit{value}$. A $\mathit{value}$ is a ground term. A configuration *covers* the cells for which it is defined, $\mathit{cover}\ C = \{p \in \mathit{loc} \mid C\ p \text{ is defined}\}$. A *substitution* is a mapping from variables to terms.

**concession, redex** If the guarding expression of a scheme holds and the preconditions are contained in the current configuration, that scheme has concession and may occur.

Let $u = n : \mathit{pre}\ \{\mathit{guard}\} \Rightarrow\ \mathit{post}$ be a fresh copy of a scheme, $\varphi$ a substitution and and C a configuration. The pair $\Delta = (u, \varphi)$ is called a *redex*, or $u$ has *concession* at C under $\varphi$, iff

- $\varphi\ \mathit{guard}$,
- ${}^\bullet\Delta \subseteq C$ the preconditions are part of the current configuration,
- $\Delta^\bullet$ is a configuration, cells can only contain one value.

where ${}^\bullet\Delta = \varphi\ \mathit{pre}$ and $\Delta^\bullet = \varphi\ \mathit{post}$ are respectively the *pre-* and the *postconditions* of $\Delta$.

**Firing, occurring** In a postcondition one can distinguish three different types of locations. First there are locations that are present in the current configuration C but not in the postconditions $\Delta^\bullet$. The values of the cells addressed by these locations remain unchanged in a rewrite step. Secondly there are locations that occur in both C as well as in $\Delta^\bullet$. The values in these locations are *updated* to the values specified by the postcondition. Finally there may be some new locations, that do not occur in C. These are added to the configuration.

Let $\Delta = (u, \varphi)$ be a redex in configuration C. The configuration $\mathit{overlap}\ \Delta^\bullet$ is the subconfiguration of C which overlaps with $\Delta^\bullet$, formally $\mathit{overlap}\ \Delta^\bullet = \{(p, C\ p) \mid p \in (\mathit{cover}\ C) \cap (\mathit{cover}\ \Delta^\bullet)\}$. Firing $u$ changes C into its successor by removing $\mathit{overlap}\ \Delta^\bullet$ from C and adding the postconditions $\Delta^\bullet$ in one atomic step:

$$C[u\rangle_\varphi C' \quad \equiv \quad C' = (C \setminus \mathit{overlap}\ \Delta^\bullet) \cup \Delta^\bullet$$

We say that transition $u$ has *occurred*.

**Script** A configuration is in *normal form* or has reached a *fixed point* if it contains no more redexes. The meaning of a script with initial configuration C is the bag of all its normal forms

$$\{C' \mid C[\rangle^* C', C \text{ in normal form}\}$$

## 3.2.4 Relating redexes in update schemes

Due to the assumptions underlying the computational model of update schemes, the relationships that may or may not hold between redexes in an update scheme are far more complicated than those encountered in a Pr/T-net.

**ordered** The configuration $\mathit{contribute}\ \Delta$ is the contribution of $\Delta$ to the change of $C$ into its successor when firing $\Delta$; $\mathit{contribute}\ \Delta = \Delta^\bullet \setminus C$. The set $\mathit{contribute}\ \Delta$ contains new locations and already present locations whose cells have different values then they have in configuration $C$.

Two redexes $\Delta$ and $\Gamma$ are *ordered*, notation $\Delta;\Gamma$, if $\mathit{contribute}\ \Delta \cap {}^\bullet\Gamma \neq \emptyset$. Since $\Gamma$ depends on changes or extensions to the configuration to be made by $\Delta$, the redex $\Gamma$ can only fire after $\Delta$.

**conflict** The *update* of a redex is the subset of the configuration that is updated when firing a redex; $\mathit{updates}\ \Delta = \mathit{overlap}\ (\mathit{contribute}\ \Delta)$. Thus the set $\mathit{updates}\ \Delta$ contains those cells in $C$ whose value will be changed, or updated, by $\Delta^\bullet$.

A redex $\Delta$ is in conflict with another redex $\Gamma$ if $\Delta$ updates cells which are in the preconditions of $\Gamma$, more formally $\mathit{updates}\ \Delta \cap {}^\bullet\Gamma \neq \emptyset$. Note that conflict is not symmetric. Firing $\Delta$ takes away concession from $\Gamma$.

**Hazard** Due to the fact that two cells cannot be written concurrently we have an interesting situation not possible in Pr/T nets, namely that the cells contributed by $\Delta$ and those contributed by $\Gamma$ overlap, more formally $\mathit{cover}\ (\mathit{contribute}\ \Delta) \cap \mathit{cover}\ (\mathit{contribute}\ \Gamma) \neq \emptyset$. Although $\Delta$ and $\Gamma$ may have concession at the same time, and firing one may leave the other in concession, they may not be fired concurrently.

A simple scheme exhibiting hazard is

$$\begin{array}{ll} \Rightarrow & X[\,3\,] \\ \Rightarrow & X[\,4\,] \end{array}$$

In Algol68 notation the above scheme would be written as `PAR (x := 3, x := 4)`.

**no order** Two redexes are independent, and can be rewritten concurrently if they both have concession but are not in conflict and there is no hazard

- $\mathit{updates}\ \Delta \cap {}^\bullet\Gamma = \emptyset \wedge \mathit{updates}\ \Gamma \cap {}^\bullet\Delta = \emptyset$ no conflict,
- $\mathit{cover}\ (\mathit{contribute}\ \Delta) \cap \mathit{cover}\ (\mathit{contribute}\ \Gamma) = \emptyset$ no hazard.

Concurrency is definitely not the same as arbitrary interleaving.

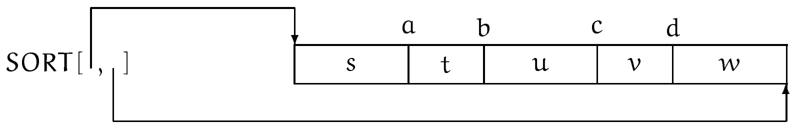
### 3.2.5 Examples

The use of update schemes to specify abstract machines will be illustrated in Chapter 5. This section presents some simple, yet elegant examples of update scheme specifications for more general programming problems, and shows how update schemes could be used to describe digital circuits. When compared with for example *Unity* [18] we find that the update scheme solutions are (nearly) free of indexitis.

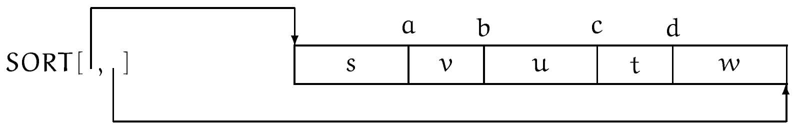
**sorting** Sorting a sequence between locations $x$ and $y$ into ascending order proceeds by non-deterministically swapping elements $t$ and $v$ that are out of order.

$$\mathrm{sort}: \quad \mathrm{SORT}[x, y] \quad x[s]a[t]b[u]c[v]d[w]y \qquad \{t > v\} \Rightarrow \quad a[v]b \quad c[t]d$$

This scheme is not only highly nondeterministic, it is also highly concurrent. Disjoint pairs of cells may be swapped in parallel. The lhs of the above scheme describes the picture:



The complete rule specifies that $t$ and $v$ should be swapped if $t > v$.



Rule $\mathrm{sort}$ has concession until the complete array between bounds $x$ and $y$ is sorted.

**maximum** A related problem is finding the maximal element of a sequence between $x$ and $y$. If there is some element $n$ between $x$ and $y$ that is bigger than the current maximum $m$, the new current maximum becomes $n$.

$$\mathrm{MAX}[x, y, m] \quad x[s]a[n]b[t]y \qquad \{n > m\} \Rightarrow \quad \mathrm{MAX}[x, y, n]$$

**semaphores** The update scheme version of semaphores is quite similar to the Pr/T-net solution. Here $\mathrm{PC}_i[pc]$ denotes the program counter of process $i$.

$$\begin{array}{llllll} V: & \mathrm{PC}_i[pc] & pc[V\ s]pc' & s[n] & \Rightarrow & \mathrm{PC}_i[pc'] \quad s[n+1] \\ P: & \mathrm{PC}_i[pc] & pc[P\ s]pc' & s[n+1] & \Rightarrow & \mathrm{PC}_i[pc'] \quad s[n] \end{array}$$

Note that the pattern $s[n+1]$ in the definition of rule P is a hidden guard, ensuring that the rule is only applicable when the value of the semaphore is at least 1.

**reachability** A graph is given by enumerating its arcs between nodes $i$ and $j$ as $[\mathrm{ARC}\ i\ j]$. The update scheme completes an initial reachability relation $[\mathrm{REACH}\ i\ b]$ where $b$ denotes that node $i$ is reachable.

$$[\mathrm{REACH}\ i\ true] \quad [\mathrm{ARC}\ i\ j] \quad [\mathrm{REACH}\ j\ false] \quad \Rightarrow \quad [\mathrm{REACH}\ j\ true]$$

**parsing** Push-down automata to recognize context-free languages are extremely easy to specify in update schemes (see also [72]). As an example we specify a recognizer for the language described by the grammar:

$$S \quad \rightarrow \quad a"; a", S.$$

Given an initial configuration $\mathrm{PARSE}[\,p\,], p[\,S\,]p'$ and $\mathrm{IN}[\,i\,], i[\,w\,]i'$, the following update scheme terminates with a configuration containing $\mathrm{PARSE}[\,p'\,]$ and $\mathrm{IN}[\,i'\,]$ iff $w$ is in the language generated by S.

$$\begin{array}{llll}
\mathrm{PARSE}[\,p\,] & p[\,S\,]p' & & \Rightarrow & \mathrm{PARSE}[\,p''\,] & p''[\,a"\,]p' & \\
\mathrm{PARSE}[\,p\,] & p[\,S\,]p' & & \Rightarrow & \mathrm{PARSE}[\,p''\,] & p''[\,a", S\,]p' & \\
\mathrm{PARSE}[\,p\,] & p[\,a"\,]p' & \mathrm{IN}[\,i\,] \quad i[\,a"\,]i' & \Rightarrow & \mathrm{PARSE}[\,p'\,] & & \mathrm{IN}[\,i'\,]
\end{array}$$

### 3.2.6 VLSI components

To describe (elementary) VLSI-components we introduce an ad-hoc abstraction mechanism for update schemes of the form $name((?loc)*, (loc!)*) : scheme*$.

A simple *wire* is defined as

$$wire(?I, O!) : I[\,x\,] \Rightarrow O[\,x\,]$$

i.e, the input of the wire is propagated to the output. An *n-type transistor* is a switch that opens if its gate is high

$$n\_trans(?G, ?S, D!) : G[\,1\,]\ S[\,x\,] \ \Rightarrow\ D[\,x\,]$$

thus if G[ 1 ] then $n\_trans(?G, ?S, D!)$ behaves as a wire $wire(?S, D!)$. Similarly a p-type transistor opens if its gate is low: $p\_trans(?G, ?S, D!) : G[\,0\,]\ S[\,x\,] \ \Rightarrow\ D[\,x\,]$.

A *Muller C-element* can be defined as

$$C(?A, ?B, C!) : A[\,x\,]\ B[\,x\,] \ \Rightarrow\ C[\,x\,]$$

If gates $A$ and $B$ carry the same value $x$, this value is propagated to output port $C$.

An *inverter* is a wire which delivers the inverted value of its input to its output.

$$\begin{array}{ll}
inv(?I, O!) : & I[\,1\,] \ \Rightarrow\ O[\,0\,] \\
 & I[\,0\,] \ \Rightarrow\ O[\,1\,]
\end{array}$$

Under the assumption that in any configuration $\mathrm{VDD}[\,1\,]$ and $\mathrm{GND}[\,0\,]$, an equivalent specification of the inverter is

$$\begin{array}{ll}
inv(?I, O!) : & I[\,1\,]\ \mathrm{GND}[\,0\,] \ \Rightarrow\ O[\,0\,] \\
 & I[\,0\,]\ \mathrm{VDD}[\,1\,] \ \Rightarrow\ O[\,1\,]
\end{array}$$

In this latter specification we recognize that $I[\,1\,]\ \mathrm{GND}[\,0\,] \ \Rightarrow\ O[\,0\,]$ equals $n\_trans($ $?I, ?GND, O!)$ and $I[\,0\,]\ \mathrm{VDD}[\,1\,] \ \Rightarrow\ O[\,1\,]$ equals $p\_trans(?I, ?VDD, O!)$. An inverter can be implemented by two transistors.

$$\begin{array}{ll}
inv(?I, O!) : & n\_trans(?I, ?GND, O!) \\
 & p\_trans(?I, ?VDD, O!)
\end{array}$$

This realization of an inverter using two complementary switches is common in relay-logic.

## 3.3 Conclusions and future work

For operations involving complicated pointer manipulations it is no superabundant luxury to have a special formalism to specify these operations. We think that update schemes are very suitable for this purpose. A disadvantage of update schemes is that they are hard to calculate with and reason about. It might be worthwhile therefore to have a look at other formalism with the same goals, such as the *structures* of Jonkers [52].

# Chapter 4

# BNF for Syntax, BMF for semantics

*Come on baby let's get out of this town*
*I got a full tank of gas with the top rolled down*
*If you won't take me with you*
*I'll go before the night is through*
*And baby you can sleep while I drive*
Melissa Etheridge

This is the first of two chapters that applies the theory developed in the previous chapters to derive compilers. The imperative language $\mathcal{C}$ is of about the same complexity as the sample languages used in most books on compiler construction. The compiler that we will derive generates realistic code suitable to be input to a conventional code generator.

## 4.1 Overview

We start by giving the syntax and a separate static and direct dynamic semantics of a simple imperative language $\mathcal{W}$. Section 4.2.4 briefly shows how static and dynamic semantics can be combined. Continuing with improving the dynamic semantics we first derive a continuation semantics; $\mathcal{W}$-programs are translated into flow-charts. Next we investigate the efficient compilation of expressions. Arithmetic expressions are translated into conventional three-address code and boolean expressions are implemented using jumping-code. The language $\mathcal{C}$ of section 4.6 extends $\mathcal{W}$ with simple procedures. We give a constructive proof that recursive functions can be implemented using a stack in §4.6.2. Finally we show how certain recursive calls may be eliminated and replaced by jumps.

## 4.2 A simple imperative language $\mathcal{W}$

As a starting point for our derivations we take a simple imperative language whose most complicated construct is the **while**-loop. In §4.6 the language $\mathcal{W}$ will be extended with non-nested procedures to yield $\mathcal{C}$. The syntax of $\mathcal{W}$ is given by the grammar

$$
\begin{array}{rcl}
P \in \mathit{program} & ::= & \textbf{prog}\ \mathit{statement} \\
S, T \in \mathit{statement} & ::= & \textbf{skip} \\
 & | & \mathit{var}\ \textbf{:=}\ \mathit{expression} \\
 & | & \mathit{statement}\ \textbf{;}\ \mathit{statement} \\
 & | & \textbf{if}\ \mathit{expression}\ \textbf{then}\ \mathit{statement}\ \textbf{else}\ \mathit{statement}\ \textbf{fi} \\
 & | & \textbf{while}\ \mathit{expression}\ \textbf{do}\ \mathit{statement}\ \textbf{od}
\end{array}
$$

The classes of arithmetic and boolean expressions are defined by the single grammar $\mathit{expression}$. Type correctness will be enforced by defining an appropriate static semantics.

$$
\begin{array}{rcl}
A, B, E, F \in \mathit{expression} & ::= & \textbf{var}\ \mathit{var} \\
 & | & \textbf{num}\ \mathit{num} \\
 & | & \textbf{true} \mid \textbf{false} \\
 & | & \mathit{expression}\ \textcircled{\scriptsize 2}\ \mathit{expression}
\end{array}
$$

The set of binary operators $\textcircled{\scriptsize 2}$ includes boolean operators $\textcircled{\scriptsize\wedge}$ such as **and** and **or**, arithmetic operators $\oplus$ like $+$ and $\times$, and relational operators $\ominus$ such as $=$ and $\geq$.

### 4.2.1 Meaning functions

The semantic functions $\mathcal{M}[\![\_]\!] \in \mathit{program} \to \mathsf{prog}$, $\mathcal{E}[\![\_]\!] \in \mathit{expression} \to \mathsf{expr}$ and $\mathcal{S}[\![\_]\!] \in \mathit{statement} \to \mathsf{stat}$ may be given by the following set of mutual recursive catamorphisms.

$$
\begin{array}{rcl}
\mathcal{M}[\![\textbf{prog}\ P]\!] & = & \mathit{program}\ \mathcal{S}[\![P]\!] \\
\mathcal{S}[\![\textbf{skip}]\!] & = & \mathit{skip} \\
\mathcal{S}[\![x\ \textbf{:=}\ E]\!] & = & \mathit{assign}\ (x, \mathcal{E}[\![E]\!]) \\
\mathcal{S}[\![S\ \textbf{;}\ T]\!] & = & \mathit{seq}\ (\mathcal{S}[\![S]\!], \mathcal{S}[\![T]\!]) \\
\mathcal{S}[\![\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!] & = & \mathit{cond}\ (\mathcal{E}[\![B]\!], \mathcal{S}[\![S]\!], \mathcal{S}[\![T]\!]) \\
\mathcal{S}[\![\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}]\!] & = & \mathit{while}\ (\mathcal{E}[\![B]\!], \mathcal{S}[\![S]\!]) \\
\mathcal{E}[\![\textbf{var}\ x]\!] & = & \mathit{var}\ x \\
\mathcal{E}[\![\textbf{num}\ n]\!] & = & \mathit{num}\ n \\
\mathcal{E}[\![\textbf{true}]\!] & = & \mathit{true} \\
\mathcal{E}[\![\textbf{false}]\!] & = & \mathit{false} \\
\mathcal{E}[\![E\ \textcircled{\scriptsize 2}\ F]\!] & = & \mathit{oper}_{\textcircled{\tiny 2}}\ (\mathcal{E}[\![E]\!], \mathcal{E}[\![F]\!])
\end{array}
$$

These equations show that the valuation functions $\mathcal{M}[\![\_]\!]$, $\mathcal{S}[\![\_]\!]$ and $\mathcal{E}[\![\_]\!]$ inductively replace each of the constructors of a given program by a corresponding compile-time semantic operation (from the algebras $prog$, $stat$ respectively $expr$). These operations should assemble the denotation of a construct from the denotations of its components in terms of operations of the run-time semantic algebra. The essence of writing a compiler is the extraction of a suitable compile-time algebra from a given run-time algebra. The recursive structure of the valuation functions is uniquely determined by the recursive structure of the abstract syntax.

### 4.2.2 Static Semantics

We will develop the static semantics $\mathcal{M}_s[\![\_]\!]$ and the dynamic semantics $\mathcal{M}_d[\![\_]\!]$ separately (omitting subscripts whenever possible) and later merge these into a single semantic function $\mathcal{M}[\![\_]\!]$. As a possible static semantics we derive and check the types of expressions and statements in a given program. It is illustrative to compare our solution to the one given in §7.2 of Reps and Teitelbaum [90].

We start by defining $\mathcal{E}[\![\_]\!] \in expression \rightarrow expr$. The type $expr$ is a map from some set of inherited attributes into a set of derived or synthesized attributes. Since we are interested in the type of expressions, one of the synthesized attributes will be of type $Type$ defined as:

$$T \in Type \quad ::= \quad Unbound \mid Num \mid Bool \mid Error$$

On $Type$ we impose the lattice structure:

$$Unbound \sqsubseteq \quad \{Num, Bool\} \quad \sqsubseteq Error$$

Expressions may contain variables so we need to maintain a compile-time environment or *symbol table* mapping variables to their type, therefore one of the inherited attributes will be of type $Symtab$:

$$\begin{array}{rcl} \eta_0\ x & = & Unbound \\ \eta[x := T]\ y & = & T, \quad\ x = y \\ & = & \eta\ y, \ \ x \neq y \end{array}$$

The empty symbol table $\eta_0$ maps every identifier to $Unbound$; binding identifier $x$ to type $T$ in $\eta$ is denoted by $\eta[x := T]$.

The type of the static semantics is $\mathcal{E}[\![\_]\!] \in expression \rightarrow (Symtab, Type) \rightarrow (Symtab, Type)$ where an inherited attribute $(\eta, T)$ consists of the types of the variables as found thus far together with the type required by the context, while the corresponding derived attribute $(\eta', T') = \mathcal{E}[\![E]\!]\ (\eta, T)$ extends $\eta$ with the types of variables occurring in $E$ and delivers the derived type of $E$. This is captured by the invariant:

$$(\eta', T') = \mathcal{E}[\![E]\!]\ (\eta, T) \quad \Rightarrow \quad \eta \sqsubseteq \eta' \ \wedge\ T \sqsubseteq T'$$

The compile-time operations for computing the static semantics are defined as follows. The type $T_x$ derived for a variable $x$ is the least type that is consistent with both the type $T$ required by the context and the type $\eta\ x$ that has already been inferred for $x$.

$$\mathcal{E}[\![\textbf{var}\ x]\!]\ (\eta, T) \quad = \quad (\eta[x := T_x], T_x)\ \mathit{where}\ T_x = (\eta\ x) \sqcup T$$

The type derived for literal constants is the smallest type that agrees with the type of the constant and the required type.

$$\begin{aligned}
\mathcal{E}[\![\textbf{true}]\!]\ (\eta, T) &= (\eta, T \sqcup \mathit{Bool}) \\
\mathcal{E}[\![\textbf{false}]\!]\ (\eta, T) &= (\eta, T \sqcup \mathit{Bool}) \\
\mathcal{E}[\![\textbf{num}\ n]\!]\ (\eta, T) &= (\eta, T \sqcup \mathit{Num})
\end{aligned}$$

For a binary operator $E \bigoplus F$ (or $A \bigotimes B$) the required type of the argument expressions $E$ and $F$ $(A,B)$ is $\mathit{Num}$ $(\mathit{Bool})$. The derived type is the lub of the derived types of the subexpressions.

$$\begin{aligned}
\mathcal{E}[\![E \bigoplus F]\!]\ (\eta, T) &= (\eta_f, T \sqcup T_f) \\
&\quad \mathit{where}\ (\eta_e, T_e) = \mathcal{E}[\![E]\!]\ (\eta, \mathit{Num}) \\
&\quad\quad\quad\quad (\eta_f, T_f) = \mathcal{E}[\![F]\!]\ (\eta_e, T_e) \\
\mathcal{E}[\![A \bigotimes B]\!]\ (\eta, T) &= (\eta_b, T \sqcup T_b) \\
&\quad \mathit{where}\ (\eta_a, T_a) = \mathcal{E}[\![A]\!]\ (\eta, \mathit{Bool}) \\
&\quad\quad\quad\quad (\eta_b, T_b) = \mathcal{E}[\![B]\!]\ (\eta_a, T_a)
\end{aligned}$$

Relational operators are overloaded, any two expressions can be related provided that they are of the same type.

$$\begin{aligned}
\mathcal{E}[\![E \ominus f]\!]\ (\eta, T) &= (\eta_f, T \sqcup (T_f \sqcap \mathit{Error}) \sqcup \mathit{Bool})) \\
&\quad \mathit{where}\ (\eta_e, T_e) = \mathcal{E}[\![E]\!]\ (\eta, \mathit{Unbound}) \\
&\quad\quad\quad\quad (\eta_f, T_f) = \mathcal{E}[\![F]\!]\ (\eta_e, T_e)
\end{aligned}$$

In giving the valuation functions, we have unfolded the respective operations $\mathit{var}, \mathit{true}$ etc. When needed in actual calculations using the fusion law, they can be recovered easily.

### Static semantics of statements

With the static semantics of expressions in hand, the static semantics of statements is relatively easy. A given symbol table $\eta$ should be updated with the types of variables occurring in the statement. Moreover any type error must be signalled. The type of the static semantics of statements is therefore $\mathcal{S}[\![\_]\!] \in \mathit{statement} \to \mathit{Symtab} \to (\mathit{Symtab}, \mathbb{B})$. If $\mathcal{S}[\![S]\!]\ \eta = (\eta', B)$ then $B$ is $\mathit{True}$ iff the statement $S$ is free of static semantic errors.

The empty statement contains no typing errors.

$$\mathcal{S}[\![\textbf{skip}]\!]\ \eta \quad = \quad (\eta, \mathit{True})$$

A type error in the expression part of an assignment is propagated upwards to the statement level.

$$\begin{array}{rcl}\mathcal{S}[\![x := E]\!]\ \eta & = & (\eta_e[x := T], T \neq \mathit{Error}) \\ & & \mathit{where}\ (\eta_e, T) = \mathcal{E}[\![E]\!]\ (\eta, \eta\ x)\end{array}$$

In composite statements, type errors occurring in components are combined and type environments are threaded through the constituent statements.

$$\begin{array}{rcl}\mathcal{S}[\![S\ ;\ T]\!]\ \eta & = & (\eta_t, c_s \wedge c_t) \\ & & \mathit{where}\ (\eta_s, c_s) = \mathcal{S}[\![S]\!]\ \eta \\ & & \qquad\quad (\eta_t, c_t) = \mathcal{S}[\![T]\!]\ \eta_s \\ \mathcal{S}[\![\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!]\ \eta & = & (\eta_t, T_b = \mathit{Bool} \wedge c_s \wedge c_t) \\ & & \mathit{where}\ (\eta_b, T_b) = \mathcal{E}[\![B]\!]\ (\eta, \mathit{Bool}) \\ & & \qquad\quad (\eta_s, c_s) = \mathcal{S}[\![S]\!]\ \eta_b \\ & & \qquad\quad (\eta_t, c_t) = \mathcal{S}[\![T]\!]\ \eta_s \\ \mathcal{S}[\![\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}]\!]\ \eta & = & (\eta_s, T_b = \mathit{Bool} \wedge c_s) \\ & & \mathit{where}\ (\eta_b, T_b) = \mathcal{E}[\![B]\!]\ (\eta, \mathit{Bool}) \\ & & \qquad\quad (\eta_s, c_s) = \mathcal{S}[\![S]\!]\ \eta_b\end{array}$$

**Static semantics of programs**

The static semantics of programs states that no static semantic error has been found and is defined by:

$$\begin{array}{rcl}\mathcal{M}[\![\_]\!] & \in & \mathit{program} \to \mathbb{B} \\ \mathcal{M}[\![\textbf{prog}\ S]\!] & = & c_s\ \mathit{where}\ (\eta, c_s) = \mathcal{S}[\![S]\!]\ \eta_0\end{array}$$

The types of identifiers appearing in S are recorded in $\eta$.

### 4.2.3 Dynamic Semantics

The initial dynamic semantics will be a standard direct semantics [95], hence the following definitions should pose no particular problems.

$$\begin{array}{rcl}\mathcal{M}[\![\_]\!] & \in & \mathit{program} \to \mathit{prog} \\ \mathcal{M}[\![\textbf{prog}\ S]\!] & = & \mathit{prgm}\ (\mathcal{S}[\![S]\!]) \\ & = & \mathcal{S}[\![S]\!]\ \eta_0 \\ \mathcal{S}[\![\_]\!] & \in & \mathit{program} \to \mathit{stat} \\ \mathcal{S}[\![\textbf{skip}]\!] & = & \mathit{skip} \\ & = & \mathit{SKIP} \\ \mathcal{S}[\![x := E]\!] & = & \mathit{assign}\ (x, \mathcal{E}[\![E]\!])\end{array}$$

$$
\begin{array}{rcl}
 & = & x := \mathcal{E}[\![E]\!] \\
\mathcal{S}[\![S \,\mathbf{;}\, T]\!] & = & seq\ (\mathcal{S}[\![S]\!], \mathcal{S}[\![T]\!]) \\
 & = & \mathcal{S}[\![S]\!]; \mathcal{S}[\![T]\!] \\
\mathcal{S}[\![\mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{fi}]\!] & = & cond\ (\mathcal{E}[\![B]\!], \mathcal{S}[\![S]\!], \mathcal{S}[\![T]\!]) \\
 & = & \mathcal{E}[\![B]\!] \to \mathcal{S}[\![S]\!] \ [\!]\ \mathcal{S}[\![T]\!] \\
\mathcal{S}[\![\mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}]\!] & = & while\ (\mathcal{E}[\![B]\!], \mathcal{S}[\![S]\!]) \\
 & = & \mu(\lambda loop. \mathcal{E}[\![B]\!] \to (\mathcal{S}[\![S]\!]\ ;\ loop)\ [\!]\ \mathrm{SKIP}) \\
\mathcal{E}[\![\_]\!] & \in & expression \to expr \\
\mathcal{E}[\![\mathbf{var}\ x]\!] & = & var\ (x) \\
 & = & \mathrm{LOOKUP}\ x \\
\mathcal{E}[\![\mathbf{num}\ n]\!] & = & num\ (n) \\
 & = & \mathrm{LITERAL}\ n \\
\mathcal{E}[\![\mathbf{true}]\!] & = & true \\
 & = & \mathrm{TRUE} = \mathrm{LITERAL}\ 1 \\
\mathcal{E}[\![\mathbf{false}]\!] & = & false \\
 & = & \mathrm{FALSE} = \mathrm{LITERAL}\ 0 \\
\mathcal{E}[\![E \circledast F]\!] & = & oper_{\circledast}\ (\mathcal{E}[\![E]\!], \mathcal{E}[\![F]\!]) \\
 & = & \mathcal{E}[\![E]\!]\ \widehat{\circledast}\ \mathcal{E}[\![F]\!]
\end{array}
$$

In the clause $\mathcal{M}[\![\mathbf{prog}\ S]\!] = \mathcal{S}[\![S]\!]\ \eta_0$, the initial environment $\eta_0$ is a run-time object.

An *environment* $\eta \in env = (var \to \mathbb{N})_\perp$ carries the dynamic values of variables appearing in an expression. Updating the environment is strict; an assignment statement $x$ **:=** $E$ updates $x$ with the *value* of $E$, thus if $\mathcal{E}[\![E]\!] = \perp$ the meaning of the statement $\mathcal{S}[\![x \mathbin{\mathbf{:=}} E]\!]$ should be $\perp$ as well. The initial environment $\eta_0$ maps every variable to $\perp$.

$$
\begin{array}{rcl}
\eta_0\ x & = & \perp \\
\eta[x := \perp] & = & \perp \\
\eta[x := v]\ y & = & v, \quad \text{if } x = y \\
 & = & \eta\ y, \ \text{if } x \neq y
\end{array}
$$

Evaluating an expression should yield a value in $\mathbb{N}$. Since expressions can contain variables, their values must be provided at run-time. Hence the denotation of expressions is a function of type $e, f, a, b \in expr = env \to \mathbb{N}$.

$$
\begin{array}{rcl}
\mathrm{LOOKUP}\ x\ \eta & = & \eta\ x \\
\mathrm{LITERAL}\ v\ \eta & = & v \\
(e\ \widehat{\circledast}\ f)\ \eta & = & e\ \eta \circledast f\ \eta
\end{array}
$$

When no confusion may arise we will just write $\circledast$ instead of $\widehat{\circledast}$.

Executing a statement modifies a given environment by updating variables with the values assigned to them in that statement $s, t, u \in stat = env \to env, prog = env$.

$$
\mathrm{SKIP}\ \eta \ = \ \eta
$$

$$
\begin{aligned}
(x := e)\ \eta &= \eta[x := e\ \eta] \\
(s\ ;\ t)\ \eta &= \mathit{strict}\ t\ (s\ \eta) \\
(b \to s\ []\ t)\ \eta &= \bot, \quad b\ \eta = \bot \\
&= s\ \eta, \quad b\ \eta = 1 \\
&= t\ \eta, \quad b\ \eta = 0
\end{aligned}
$$

where $\mathit{strict}\ f\ \bot = \bot$ and $\mathit{strict}\ f\ x = f\ x$ if $x \neq \bot$. Sequential composition ; is defined by strict composition as we want $\bot\ ;\ t = \bot$ regardless of the value of $t$.

**Example** As an example take a program for computing the factorial for $n \geq 1$:

```
fac := num 1 ;
while (var n ≥ num 1)
do
   fac := (var n × var fac) ;
   n := (var n − num 1)
od
```

The denotation of this simple program is the recursive function

```
factorial  =  fac := (LITERAL 1) ;
              μ(λloop.(LOOKUP n ≥ LITERAL 1) →
                   (fac := (LOOKUP n × LOOKUP fac) ;
                   n := (LOOKUP n − LITERAL 1) ; loop)
              [] SKIP)
```

### 4.2.4 Merging static and dynamic semantics

Thus far static and dynamic semantics were defined separately, e.g., for expressions:

$$
\begin{aligned}
\mathcal{E}_s[\![\_]\!] &\in \mathit{expression} \to (\mathit{Symtab}||\mathit{Type}) \to (\mathit{Symtab}||\mathit{Type}) \\
\mathcal{E}_s[\![\_]\!] &= (\![\mathit{var}_s, \mathit{true}_s, \mathit{false}_s, \mathit{num}_s, \mathit{oper}_s]\!) \\
\mathcal{E}_d[\![\_]\!] &\in \mathit{expression} \to \mathit{expr} \\
\mathcal{E}_d[\![\_]\!] &= (\![\mathit{var}_d, \mathit{true}_d, \mathit{false}_d, \mathit{num}_d, \mathit{oper}_d]\!)
\end{aligned}
$$

We want to combine these two functions into a single function:

$$
\begin{aligned}
\mathcal{E}[\![\_]\!] &\in \mathit{expression} \to (\mathit{Symtab}||\mathit{Type}) \to (\mathit{Symtab}||\mathit{Type}||\mathit{expr}) \\
\mathcal{E}[\![\_]\!] &= (\![\mathit{var}, \mathit{true}, \mathit{false}, \mathit{num}, \mathit{plus}, \mathit{oper}]\!)
\end{aligned}
$$

From the given semantic equations, it is easy to recover the various compile-time operations such as $\mathit{oper}_s$ and $\mathit{oper}_d$:

$$
\begin{aligned}
\mathit{oper}_s\ (e, f) &= \lambda(\eta, T).(\eta_f, T \sqcup T_f) \\
&\quad \mathit{where}\ (\eta_e, T_e) = e\ (\eta, \mathit{Num}); (\eta_f, T_f) = f\ (\eta_e, T_e); \\
\mathit{oper}_d\ (e, f) &= (E \mathbin{\widehat{\oplus}} F)\ \mathit{where}\ E = e; F = f
\end{aligned}
$$

Combining these rules into a single one using the *Attribute Grammar Tupling* rule (AGF) yields:

$$\begin{aligned}\mathit{plus}\ (e, f) &= \lambda(\eta, T).(\eta_f, T \sqcup T_f, E \mathbin{\widehat{\oplus}} F)\\ &\quad \mathit{where}\ (\eta_e, T_e, E) = e\ (\eta, \mathit{Num})\\ &\qquad\qquad (\eta_f, T_f, F) = f\ (\eta_e, T_f)\end{aligned}$$

Similarly the other equations can be combined. It is not the right time and place here to digress on the advantage of modular descriptions in general so we conclude that our proposed approach has good support for modularity.

## 4.3 Continuation Semantics

In his thesis, de Bruin [25] experimented with continuation semantics for defining language concepts such as jumps, backtracking and dynamic process networks. Explicit manipulation of control, i.e. the evaluation order of a program's constructs, in the form of continuations will play a central rôle in our work as well. Invariably every efficiency improving transformation is aimed at making explicit an otherwise implicit evaluation order by the introduction of an additional continuation.

The first such continuation introduction makes the control-flow in sequencing of statements $\acute{s}; \acute{t}$ explicit. A statement denotation $\acute{s} \in \mathit{env} \to \mathit{env}$ should be implemented as a function $\grave{s} = C\ \acute{s} \in (\mathit{env} \to \mathit{env}) \to (\mathit{env} \to \mathit{env})$ such that $C\ \acute{s}$ takes its continuation as an explicit argument. Type considerations strongly suggest to define $C$ as

$$C\ \acute{s}\ \acute{t} \quad = \quad \acute{s}\ ; \acute{t}$$

The left-inverse $A$ of $C$ takes a concrete $\grave{s}$ that expects a continuation back into an abstract one $A\ \grave{s}$ that does not.

$$\begin{aligned}&A\ \grave{s}\\ =\quad &\quad \grave{s} = C\ \acute{s}\\ &A\ (C\ \acute{s})\\ =\quad &\quad \text{wish}\\ &\acute{s}\\ =\quad &\quad \text{aim at folding } C\\ &\acute{s}; \mathrm{id}\\ =\quad &\quad \text{fold}\\ &C\ \acute{s}\ \mathrm{id}\\ =\quad &\quad C\ \acute{s} = \grave{s}\\ &\grave{s}\ \mathrm{id}\end{aligned}$$

Thus $\grave{s} = C\ \acute{s} \Rightarrow \acute{s} = A\ \grave{s}$, i.e., we have found that $A\ \grave{s} = \grave{s}\ \mathit{id}$ is a left-inverse of $C$. Using the fact that $A \circ C = \mathit{id} \in \mathit{env} \to \mathit{env}$ the previous direct semantics can be turned into a continuation semantics.

$$
\begin{array}{ll}
 & \grave{\mathcal{M}}[\![\textbf{prog}\ S]\!] \\
= & \quad \text{demand} \\
 & \acute{\mathcal{M}}[\![\textbf{prog}\ S]\!] \\
= & \quad \text{unfold} \\
 & \acute{\mathcal{S}}[\![S]\!]\ \eta_0 \\
= & \quad A \circ C = \mathit{id} \\
 & (A \circ C \circ\ \acute{\mathcal{S}}[\![S]\!])\ \eta_0 \\
= & \quad \text{unfold} \\
 & C\ \acute{\mathcal{S}}[\![S]\!]\ \mathit{id}\ \eta_0 \\
= & \quad \text{assume fusion: } C \circ \acute{\mathcal{S}}[\![\_]\!] = \grave{\mathcal{S}}[\![\_]\!] \\
 & \grave{\mathcal{S}}[\![S]\!]\ \mathit{id}\ \eta_0
\end{array}
$$

The proof of the postulate $C \circ \acute{\mathcal{S}}[\![\_]\!] = \grave{\mathcal{S}}[\![\_]\!]$ using the fusion law determines new compile-time operations (and thereby new run-time operations) that satisfy the premisses of the fusion law.

$$
\begin{array}{rcl}
C \circ \acute{\mathcal{S}}[\![\_]\!] = \grave{\mathcal{S}}[\![\_]\!] & \Leftarrow & C\ \mathit{sk\acute{i}p} = \mathit{sk\grave{i}p}\ \wedge \\
 & & C \circ \mathit{ass\acute{i}gn} = \mathit{ass\grave{i}gn} \circ \mathit{id}||\mathit{id}\ \wedge \\
 & & C \circ \mathit{s\acute{e}q} = \mathit{s\grave{e}q} \circ C||C\ \wedge \\
 & & C \circ \mathit{c\acute{o}nd} = \mathit{c\grave{o}nd} \circ \mathit{id}||C||C\ \wedge \\
 & & C \circ \mathit{wh\acute{i}le} = \mathit{wh\grave{i}le} \circ \mathit{id}||C
\end{array}
$$

Rewritten with bound variables, but omiting quantifiers this theorem reads

$$
\begin{array}{rcl}
C\ \acute{\mathcal{S}}[\![S]\!]\ s = \grave{\mathcal{S}}[\![S]\!]\ s & \Leftarrow & C\ \mathit{sk\acute{i}p}\ s = \mathit{sk\grave{i}p}\ s\ \wedge \\
 & & C\ (\mathit{ass\acute{i}gn}\ (x,e))\ s = \mathit{ass\grave{i}gn}\ (x,e)\ s\ \wedge \\
 & & C\ (\mathit{s\acute{e}q}\ (s,t)\ u = \mathit{s\grave{e}q}\ (C\ s, C\ t)\ u\ \wedge \\
 & & C\ (\mathit{c\acute{o}nd}\ (b,s,t))\ u = \mathit{c\grave{o}nd}\ (b, C\ s, C\ t)\ u\ \wedge \\
 & & C\ (\mathit{wh\acute{i}le}\ (b,s))\ t = \mathit{wh\grave{i}le}\ (b, C\ s)\ t
\end{array}
$$

This version reflects that continuations are compile-time objects and may thus be used to derive a more efficient compiler if partial evaluation is possible.

One particular branch in the resulting proof may be summarized as

$$
\begin{array}{ll}
 & \acute{\mathcal{M}}[\![\_]\!] = \grave{\mathcal{M}}[\![\_]\!] \\
\Leftarrow & \quad A \circ C = \mathit{id} \\
 & C\ \acute{\mathcal{S}}[\![S]\!]\ s = \grave{\mathcal{S}}[\![S]\!]\ s \\
\Leftarrow & \quad \text{fusion}
\end{array}
$$

$$\begin{aligned}
& \ldots \wedge\ C\ (\mathit{ass\acute{\imath}gn}\ (x,e))\ s = \mathit{ass\grave{\imath}gn}\ (x,e)\ s\ \wedge \ldots \\
\Leftarrow \quad & \quad \text{unfold C and } \mathit{ass\acute{\imath}gn}\text{, extract } \mathit{ass\grave{\imath}gn} \\
& \ldots \wedge\ \mathit{ass\grave{\imath}gn}\ (x,e)\ s = (x :\acute{=} e)\ s\ \wedge \ldots \\
\Leftarrow \quad & \quad \text{evaluate } :\acute{=}\text{, synthesize } :\grave{=} \\
& \ldots \wedge\ (x :\grave{=} e)\ s\ \eta = s\ \eta[x := e\ \eta]\ \wedge \ldots
\end{aligned}$$

**skip** We try to find $\grave{\mathcal{S}}[\![\textbf{skip}]\!]\ s$ under the assumption that $s$ is available as a compile-time entity.

$$\begin{aligned}
& C\ \mathit{sk\acute{\imath}p}\ s \\
= \quad & \quad \text{unfold} \\
& (\mathrm{SK\acute{I}P}\ ;\ s) \\
= \quad & \quad \text{evaluate} \\
& s \\
= \quad & \quad \text{extract} \\
& \mathit{sk\grave{\imath}p}\ s
\end{aligned}$$

Indeed $C\ \mathit{sk\acute{\imath}p} = \mathit{sk\grave{\imath}p}$, note that all instances of SKIP are compiled away; $\mathit{sk\grave{\imath}p}\ s = s$. If $s$ was not available at compile-time we could only have synthesized a new run-time operation $\mathrm{SK\grave{I}P}\ s\ \eta = s\ \eta$.

**Assignments** give rise to a new run-time and a new compile-time operation.

$$\begin{aligned}
& C\ (\mathit{ass\acute{\imath}gn}\ (x,e))\ s \\
= \quad & \quad \text{unfold} \\
& x :\acute{=} e\ ;\ s \\
= \quad & \quad \text{abutting calculation} \\
& (x :\grave{=} e)\ s \\
= \quad & \quad \text{extract} \\
& \mathit{ass\grave{\imath}gn}\ (x,e)\ s\ \eta
\end{aligned}$$

$$\begin{aligned}
& (x :\acute{=} e\ ;\ s)\ \eta \\
= \quad & \quad \text{evaluate} \\
& s\ (\eta[x := e\ \eta]) \\
= \quad & \quad \text{synthesize} \\
& (x :\grave{=} e)\ s\ \eta
\end{aligned}$$

Thus we have shown $C \circ \mathit{ass\acute{\imath}gn} = \mathit{ass\grave{\imath}gn} \circ \mathit{id}||\mathit{id}$ and thereby we synthesized a new run-time operation $(x :\grave{=} e)\ s\ \eta = s\ \eta[x := e\ \eta]$ and extracted a new compile-time operation $\mathit{ass\grave{\imath}gn}\ (x,e)\ s = (x :\grave{=} e)\ s$.

**sequential composition and conditionals** are nearly as easy as $\mathit{skip}$ and $\mathit{assign}$.

$$\begin{array}{ll} & C\ (\acute{seq}\ (s,t))\ u \\ = & \quad \text{unfold} \\ & s;t;u \\ = & \quad \text{fold twice} \\ & C\ s\ (C\ t\ u) \\ = & \quad \text{extract} \\ & \grave{seq}\ (C\ s, C\ t)\ u \end{array}$$

$$\begin{array}{ll} & C\ (\acute{cond}\ (b,s,t))\ u \\ = & \quad \text{unfold} \\ & (b \rightarrow s \ [\!] \ t);u \\ = & \quad \text{property conditional} \\ & b \rightarrow (s;u) \ [\!] \ (t;u) \\ = & \quad \text{fold twice} \\ & b \rightarrow (C\ s\ u) \ [\!] \ (C\ t\ u) \\ = & \quad \text{extract} \\ & \grave{cond}\ (b, C\ s, C\ t)\ u \end{array}$$

**while-loops** The most interesting case is the **while**-loop where recursion is involved. The following lemma, which is easily proved by fixed point induction on $P(f,g) \equiv f;u = g$ using $f = \lambda l.(b \rightarrow (s\,;l) \ [\!] \ t)$ and $g = \lambda l.b \rightarrow (s\,;l) \ [\!] \ (t\,;u)$,

$$\mu(\lambda loop.(b \rightarrow (s\,;loop) \ [\!] \ t))\,;u \quad = \quad \mu(\lambda loop.b \rightarrow (s\,;loop) \ [\!] \ (t\,;u)) \qquad (4.1)$$

intuitively says that loops are tail-recursive, and hence can be realized by a flow-chart program.

$$\begin{array}{ll} & C\ (\acute{while}\ (b,s))\ t \\ = & \quad \text{unfold} \\ & \mu(\lambda loop.(b \rightarrow (s\,;loop) \ [\!] \ SKIP))\,;t \\ = & \quad \text{(4.1) and evaluate} \\ & \mu(\lambda loop.b \rightarrow (s\,;loop) \ [\!] \ t) \\ = & \quad \text{fold} \\ & \mu(\lambda loop.b \rightarrow (C\ s\ loop) \ [\!] \ t) \\ = & \quad \text{extract} \\ & \grave{while}\ (b, C\ s)\ t \end{array}$$

Applying the fusion law and unfolding the compile-time operations $skip$, $assign$ etc, results in a continuation semantics for $\mathcal{W}$.

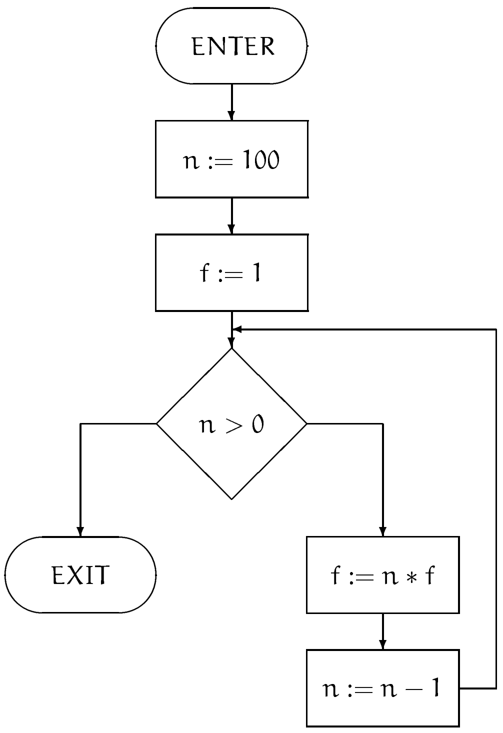
$$\begin{array}{rcl} \mathcal{M}[\![\textbf{prog}\ S]\!] & = & \mathcal{S}[\![S]\!]\ id\ \eta_0 \\ \\ \mathcal{S}[\![\textbf{skip}]\!]\ s & = & s \\ \mathcal{S}[\![x := E]\!]\ s & = & x := \mathcal{E}[\![E]\!]\ s \\ \mathcal{S}[\![S\ ;\ T]\!]\ u & = & \mathcal{S}[\![S]\!]\ (\mathcal{S}[\![T]\!]\ u) \\ \mathcal{S}[\![\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!]\ u & = & \mathcal{E}[\![B]\!] \rightarrow (\mathcal{S}[\![S]\!]\ u) \ [\!] \ (\mathcal{S}[\![T]\!]\ u) \\ \mathcal{S}[\![\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}]\!]\ t & = & \mu(\lambda loop.\mathcal{E}[\![B]\!] \rightarrow (\mathcal{S}[\![S]\!]\ loop) \ [\!] \ t) \end{array}$$

$$\begin{aligned}
\mathcal{E}[\![\textbf{var}\ x]\!] &= \text{LOOKUP}\ x\\
\mathcal{E}[\![\textbf{num}\ n]\!] &= \text{LITERAL}\ n\\
\mathcal{E}[\![\textbf{true}]\!] &= \text{TRUE}\\
\mathcal{E}[\![\textbf{false}]\!] &= \text{FALSE}\\
\mathcal{E}[\![E ② F]\!] &= \mathcal{E}[\![E]\!]\ \widehat{②}\ \mathcal{E}[\![F]\!]
\end{aligned}$$

The clauses for expressions remain yet unchanged.

Our continuation semantics for $\mathcal{W}$ is the same as the semantics given by de Bruin [25] and Schmidt [95] for a similar language, except that Schmidt swaps the continuation and environment arguments. He makes no distinction between static and dynamic arguments.

**example** The current semantics translates programs into flow-charts [98]. The program for computing factorials as given earlier, with $n$ initialized to 100 for example, would compile into



The well known idea of cutting loops and associating recursive equations with a flow-chart [66] can be applied here as well. Cut-points become 'labels' and all but one of the arcs pointing at such labels become GOTO's.

$$\text{GOTO}\ label\ t \;=\; label$$

$$\begin{aligned} \mathrm{IF}_{false}\ b\ s\ t &= b \rightarrow t \ [\!]\ s \\ \mathrm{EXIT}\ s\ \eta &= \eta \end{aligned}$$

Using these extra instructions the denotation for the factorial program reminds of machine code:

$$\begin{aligned} factorial &= (fac := (\mathrm{LITERAL}\ 1))\ loop \\ loop &= (\mathrm{IF}_{false}\ (\mathrm{LOOKUP}\ n \geq \mathrm{LITERAL}\ 1)\ done \circ \\ &\quad\ fac := (\mathrm{LOOKUP}\ n \times \mathrm{LOOKUP}\ fac) \circ \\ &\quad\ n := (\mathrm{LOOKUP}\ n - \mathrm{LITERAL}\ 1) \circ \\ &\quad\ \mathrm{GOTO}\ loop)\ \bot \\ done &= \mathrm{EXIT}\ \bot \end{aligned}$$

The translation scheme for flow-of-control statements can be modified easily to produce code with labels and jumps akin to code generated by conventional compliers.

$$\begin{aligned} \mathcal{S}[\![\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!]\ u &= \mathrm{IF}_{false}\ \mathcal{E}[\![B]\!]\ t\ (\mathcal{S}[\![S]\!]\ (\mathrm{GOTO}\ u\ \bot)) \\ &\quad\ where\ t = (\mathcal{S}[\![T]\!]\ u) \\ \mathcal{S}[\![\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}]\!]\ t &= loop \\ &\quad\ where\ loop = \mathrm{IF}_{false}\ \mathcal{E}[\![B]\!]\ t\ (\mathcal{S}[\![S]\!]\ (\mathrm{GOTO}\ loop\ \bot)) \end{aligned}$$

## 4.4 Expression Continuations

The next goal is to make control-flow in the evaluation of expressions explicit. Looking at $\mathcal{E}[\![E ② F]\!]\ \eta = \mathcal{E}[\![E]\!]\ \eta ② \mathcal{E}[\![F]\!]\ \eta$, we see that the order of evaluating the arguments of ② is not specified. For an actual implementation some order must be chosen, and subsequently intermediate values have to be stored. Usually a stack is introduced for this purpose. Explicit naming of intermediate values is not only easier to derive, it also gives better code for modern load-store RISC architectures such as the *Motorola 88.000* [79]. Pettersson [87] also generates three-address code (but from an already given) continuation-style semantics. His correctness proofs are very informal, especially when dealing with functions. He concludes for example that a callee-saves protocol is needed for function calls. In our opinion this is not the case at all. The choice between callee-saves and caller-saves can (and should) be left to the final code generator.

Explicit control-flow can be introduced into the evaluation of expressions by lifting the order of evaluation subexpressions to the statement level.

$$\mathcal{S}[\![x := E ② F]\!] = \mathcal{S}[\![y := E\ ; z := F\ ; x := y ② z]\!] \qquad \text{(ExprSimpl)}$$

where $y$ and $z$ are fresh variables.

This suggest that expressions should be turned into statements by means of the transformation $E \in expr \rightarrow (var || stat) \rightarrow stat$; the type of $E$ leaves little choice but taking

$$E\ e\ (x, s) = (x := e)\ s \qquad (4.2)$$

Although $E$ may generate unnecessary assignments such as in $E\ \mathcal{E}[\![\textbf{var}\ x + \textbf{var}\ y]\!]\ (z,s)$, it is often simpler to eliminate these in subsequent compilation phases than to complicate the transformation to deal with these special cases. If we use for example graph coloring [5] to map identifiers into machine registers, redundant moves $x := \text{LOOKUP}\ y$ are eliminated for free.

Instantiating the fusion law yields the following condition under which the optimization $E$ may be incorporated into the semantics.

$$
\begin{array}{lll}
\grave{\mathcal{S}}[\![\_]\!] = \acute{\mathcal{S}}[\![\_]\!] \ \wedge\ E \circ \acute{\mathcal{E}}[\![\_]\!] = \grave{\mathcal{E}}[\![\_]\!] & \Leftarrow & \acute{cond} = \grave{cond} \circ E \| id \| id \ \wedge \\
 & & \acute{while} = \grave{while} \circ E \| id \ \wedge \\
 & & \acute{assign} = \grave{assign} \circ E \| id \ \wedge \\
 & & E \circ \acute{var} = \grave{var} \ \wedge\ E \circ \acute{num} = \grave{num} \ \wedge \\
 & & E\ \acute{true} = \grave{true} \ \wedge\ E\ \acute{false} = \grave{false} \ \wedge \\
 & & E \circ \acute{oper} = \grave{oper} \circ E \| E
\end{array}
$$

**variables and constants** New compile-time (and run-time) operations are extracted (synthesized) by calculating. As usual we start with the simplest cases.

$$
\begin{array}{ll}
 & E\ (\acute{var}\ y)\ (x,s) \\
= & \quad \text{unfold} \\
 & (x := \text{LOOKUP}\ y)\ s \\
= & \quad \text{extract} \\
 & \grave{var}\ y\ (x,s)
\end{array}
\qquad
\begin{array}{ll}
 & E\ (\acute{num}\ n)\ (x,s) \\
= & \quad \text{unfold} \\
 & x := \ (\text{LITERAL}\ n)\ s \\
= & \quad \text{extract} \\
 & \grave{num}\ n\ (x,s)
\end{array}
$$

This gives new compile-time operations $\grave{var}\ y\ (x,s) = (x := \text{LOOKUP}\ y)\ s$ and $\grave{num}\ n\ (x,s) = (x := \text{LITERAL}\ n)\ s$.

**binary operators** Improving the compilation of binary operators is the reason why we do these calculations in the first place. The calculation is driving towards folding $E$ on the subexpressions $e$ and $f$, thereby using the observation as a heuristic.

$$
\begin{array}{ll}
 & E\ (\acute{oper}_{②}\ (e,f))\ (x,s) \\
= & \quad \text{unfold} \\
 & x := (e\ \widehat{②}\ f)\ s \\
= & \quad \text{observation (ExprSimpl), } y \text{ and } z \text{ fresh variables} \\
 & y := e\ (z := f\ (x := (\text{LOOKUP}\ y\ \widehat{②}\ \text{LOOKUP}\ z)\ s)) \\
= & \quad \text{fold twice} \\
 & E\ e\ (y, E\ f\ (z, x := (\text{LOOKUP}\ y\ \widehat{②}\ \text{LOOKUP}\ z)\ s)) \\
= & \quad \text{synthesize} \\
 & E\ e\ (y, E\ f\ (z, x := (y\ ②\ z)\ s)) \\
= & \quad \text{extract}
\end{array}
$$

$$o\grave{p}er\ (E\ e, E\ f)\ (x, s)$$

This yields $x := (y \textcircled{2} z)\ s\ \eta = s\ \eta[x := \eta\ y \textcircled{2} \eta\ z]$ and $o\grave{p}er_{\textcircled{2}}\ (e, f)\ (x, s) = e\ (y, f\ (z, x := (y \textcircled{2} z)\ s))$.

**statements** In order to apply fusion, the optimization E should be embedded into the meaning of statements. The key observation is the following equivalence that makes the evaluation order in a conditional statement explicit.

$$\mathcal{S}[\![\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!] \quad = \quad \mathcal{S}[\![x := B\ \textbf{;}\ \textbf{if}\ (\textbf{var}\ x)\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!]$$

With this insight the derivation of the new compilation scheme for conditionals is simple enough.

$$\begin{array}{ll}
 & c\acute{o}nd\ (b, s, t)\ u \\
= & \quad \text{unfold} \\
 & b \to (s\ u)\ [\!]\ (t\ u) \\
= & \quad \text{eureka} \\
 & x := b\ (\text{LOOKUP}\ x \to (s\ u)\ [\!]\ (t\ u)) \\
= & \quad \text{fold} \\
 & E\ b\ (x, (\text{LOOKUP}\ x \to (s\ u)\ [\!]\ (t\ u))) \\
= & \quad \text{extract} \\
 & c\grave{o}nd\ (E\ b, s, t)\ u
\end{array}$$

Similarly we find $ass\grave{i}gn\ (x, e)\ s = e\ (x, s)$ and $wh\grave{i}le\ (b, s)\ t = \mu(\lambda l.b\ (x, (\text{LOOKUP}\ x \to (s\ l)\ [\!]\ t)))$ (no fixed point induction needed).

Refering to the fusion law yields the new semantics:

$$\begin{array}{rcl}
\mathcal{M}[\![\textbf{prog}\ S]\!] & = & \mathcal{S}[\![S]\!]\ \text{id}\ \eta_0 \\
\\
\mathcal{S}[\![\textbf{skip}]\!]\ s & = & s \\
\mathcal{S}[\![x\ \textbf{:=}\ E]\!]\ s & = & \mathcal{E}[\![E]\!]\ (x, s) \\
\mathcal{S}[\![\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!]\ u & = & \mathcal{E}[\![B]\!]\ (x, \text{LOOKUP}\ x \to (\mathcal{S}[\![S]\!]\ u)\ [\!]\ (\mathcal{S}[\![S]\!]\ u)) \\
\mathcal{S}[\![\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}]\!]\ t & = & \mu(\lambda loop.\mathcal{E}[\![B]\!]\ (x, \text{LOOKUP}\ x \to (\mathcal{S}[\![S]\!]\ loop)\ [\!]\ t)) \\
\\
\mathcal{E}[\![\textbf{var}\ y]\!]\ (x, s) & = & (x := \text{LOOKUP}\ y)\ s \\
\mathcal{E}[\![\textbf{num}\ n]\!]\ (x, s) & = & (x := \text{LITERAL}\ n)\ s \\
\mathcal{E}[\![\textbf{true}]\!]\ (x, s) & = & (x := \text{TRUE})\ s \\
\mathcal{E}[\![\textbf{false}]\!]\ (x, s) & = & (x := \text{FALSE})\ s \\
\mathcal{E}[\![E \textcircled{2} F]\!]\ (x, s) & = & \mathcal{E}[\![E]\!]\ (y, \mathcal{E}[\![F]\!]\ (z, x := (y \textcircled{2} z)\ s))
\end{array}$$

The new set of run-time operations is:

$$\begin{aligned}
(x := \mathrm{LOOKUP}\ y)\ s\ \eta &= s\ \eta[x := \eta\ y] \\
(x := \mathrm{LITERAL}\ n)\ s\ \eta &= s\ \eta[x := n] \\
x := (y ② z)\ s\ \eta &= s\ \eta[x := \eta\ y ② \eta\ z]
\end{aligned}$$

The full continuation version, using labels and GOTO's, of our example script looks like:

$$\begin{aligned}
\mathit{factorial} &= (r := \mathrm{LITERAL}\ 1 \circ \\
&\quad\ \mathit{fac} := r)\ \mathit{loop} \\
\mathit{loop} &= (r0 := \mathrm{LITERAL}\ 1 \circ \\
&\quad\ r1 := \mathrm{LOOKUP}\ n \circ \\
&\quad\ r2 := (r1 \geq r0) \circ \\
&\quad\ \mathrm{IF}_{\mathit{false}}\ r2\ \mathit{done} \circ \\
&\quad\ r3 := \mathrm{LOOKUP}\ \mathit{fac} \circ \\
&\quad\ r4 := \mathrm{LOOKUP}\ n \circ \\
&\quad\ \mathit{fac} := r4 \times r3 \circ \\
&\quad\ r5 := \mathrm{LITERAL}\ 1 \circ \\
&\quad\ r6 := \mathrm{LOOKUP}\ n \circ \\
&\quad\ n := r6 - r5 \circ \\
&\quad\ \mathrm{GOTO}\ \mathit{loop})\ \bot \\
\mathit{done} &= \mathrm{EXIT}\ \bot
\end{aligned}$$

where $\mathrm{IF}_{\mathit{false}}\ x\ s\ t = \mathrm{LOOKUP}\ x \rightarrow t\ [\!]\ s$.

## 4.5 Short Circuit Evaluation

Most programming languages, with Algol68 and Pascal being exceptions to the rule, specify *short circuit* evaluation of boolean expressions. The C book [53] for example says:

> "Unlike &, && guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is equal to 0, the value of the expression is 0. Otherwise the right operand is evaluated, and if it is equal to 0, the expression's value is 0, otherwise 1."

Our short circuit implementation of boolean expressions is motivated by the equivalences

$$\begin{aligned}
\mathcal{S}[\![x := B]\!] &= \mathcal{S}[\![\textbf{if}\ B\ \textbf{then}\ x := \textbf{true else}\ x := \textbf{false fi}]\!] \\
\mathcal{S}[\![\textbf{if}\ (A\ \textbf{or}\ B)\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!] &= \mathcal{S}[\![\textbf{if}\ A\ \textbf{then}\ S\ \textbf{else}\ (\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi})\ \textbf{fi}]\!] \\
\mathcal{S}[\![\textbf{if}\ (A\ \textbf{and}\ B)\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!] &= \mathcal{S}[\![\textbf{if}\ A\ \textbf{then}\ (\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi})\ \textbf{else}\ T\ \textbf{fi}]\!]
\end{aligned}$$

This allows short circuit evaluation of boolean expressions to be implemented with very little boolean 'values' ever existing.

If short circuit code is required, boolean connectives have to be translated differently from arithmetical operators. Therefore the syntax of $\mathcal{W}$ is changed to distinguish between arithmetic and boolean expressions. It is possible to introduce short circuit evaluation without changing the syntax, however these solutions are essentially the same as the proposed one but much more awkard to deal with. We will return to this point point in section §4.5.1

$$\begin{array}{rcl}
P \in \mathit{program} & ::= & \textbf{prog}\ \mathit{statement} \\
S, T \in \mathit{statement} & ::= & \textbf{skip} \\
 & | & \mathit{var}\ \textbf{:=}\ \mathit{expression} \\
 & | & \mathit{statement}\ \textbf{;}\ \mathit{statement} \\
 & | & \textbf{if}\ \mathit{boolean}\ \textbf{then}\ \mathit{statement}\ \textbf{else}\ \mathit{statement}\ \textbf{fi} \\
 & | & \textbf{while}\ \mathit{boolean}\ \textbf{do}\ \mathit{statement}\ \textbf{od} \\
E, F \in \mathit{expression} & ::= & \textbf{var}\ \mathit{var} \\
 & | & \textbf{num}\ \mathit{num} \\
 & | & \mathit{expression} \bigoplus \mathit{expression} \\
 & | & \textbf{bool}\ \mathit{boolean} \\
A, B \in \mathit{boolean} & ::= & \textbf{var}\ \mathit{var} \\
 & | & \textbf{true} \mid \textbf{false} \\
 & | & \mathit{expression}\ ⊜\ \mathit{expression} \\
 & | & \mathit{boolean}\ \circledwedge\ \mathit{boolean}
\end{array}$$

Formally short circuit evaluation of boolean expressions can be introduced by simultaneous application of the transformations:

$$\begin{array}{rcl}
B\ b\ (s,t) & = & b \rightarrow s \mathbin{[\!]} t \\
E\ e\ (x,s) & = & x := e\ s
\end{array}$$

First we will derive the new semantic operations for boolean expressions. The generated code for a variable dynamically chooses a branch to continue because the value of variable $x$ is not known until at run-time.

$$\begin{array}{rl}
 & B\ (\acute{var}\ x)\ (s,t) \\
= & (\mathrm{LOOKUP}\ x) \rightarrow s \mathbin{[\!]} t \\
= & \grave{var}\ x\ (s,t)
\end{array}$$

Atomic boolean expressions are eliminated at compile-time, since the pair of continuations $(s,t)$ is a compile-time object.

$$\begin{array}{rl}
 & B\ \acute{true}\ (s,t) \\
= & \mathrm{TRUE} \rightarrow s \mathbin{[\!]} t \\
= & s \\
= & \grave{true}\ (s,t)
\end{array}
\qquad\qquad
\begin{array}{rl}
 & B\ \acute{false}\ (s,t) \\
= & \mathrm{FALSE} \rightarrow s \mathbin{[\!]} t \\
= & t \\
= & \grave{false}\ (s,t)
\end{array}$$

Complex boolean expressions are reduced to a nest of conditionals as suggested by the above observations.

$$\begin{aligned}
& \mathrm{B}\ (\text{ánd}\ (a,b))\ (s,t) \\
= {}& (a\ \mathrm{AND}\ b) \to s\ [\,]\ t \\
= {}& \quad \text{observation} \\
& a \to (b \to s\ [\,]\ t)\ [\,]\ t \\
= {}& \mathrm{B}\ a\ (\mathrm{B}\ b\ (s,t),t) \\
= {}& \text{ànd}\ (\mathrm{B}\ a,\mathrm{B}\ b)\ (s,t)
\end{aligned}
\qquad\qquad
\begin{aligned}
& \mathrm{B}\ (\text{ór}\ (a,b))\ (s,t) \\
= {}& (a\ \mathrm{OR}\ b) \to s\ [\,]\ t \\
= {}& \quad \text{observation} \\
& a \to s\ [\,]\ (b \to s\ [\,]\ t) \\
= {}& \mathrm{B}\ a\ (s,\mathrm{B}\ b\ (s,t)) \\
= {}& \text{òr}\ (\mathrm{B}\ a,\mathrm{B}\ b))\ (s,t)
\end{aligned}$$

Relational expressions cannot be encoded by means of control-flow, and thus require somewhat more work.

$$\begin{aligned}
& \mathrm{B}\ (\text{eq́ual}\ (e,f))\ (s,t) \\
= {}& (e \text{ ⊜ } f) \to s\ [\,]\ t \\
= {}& x := (e \text{ ⊜ } f)\ (\mathrm{LOOKUP}\ x \to s\ [\,]\ t) \\
= {}& y := e\ (z := f\ (x := y \text{ ⊜ } z\ (\mathrm{LOOKUP}\ x \to s\ [\,]\ t))) \\
= {}& \mathrm{E}\ e\ (y,\mathrm{E}\ f\ (z,x := y \text{ ⊜ } x\ (\mathrm{LOOKUP}\ x \to s\ [\,]\ t))) \\
= {}& \text{eq̀ual}\ (\mathrm{E}\ e,\mathrm{E}\ f)\ (s,t)
\end{aligned}$$

Boolean expressions form the interface between E and B, it is assumed that $\text{bo\'ol} = \mathit{id}$.

$$\begin{aligned}
& \mathrm{E}\ (\text{bo\'ol}\ b)\ (x,s) \\
= {}& (x := b)\ s \\
= {}& b \to (x := \mathrm{TRUE}\ s)\ [\,]\ (x := \mathrm{FALSE}\ s) \\
= {}& \mathrm{B}\ b\ (x := \mathrm{TRUE}\ s, x := \mathrm{FALSE}\ s) \\
= {}& \text{bo\`ol}\ (\mathrm{B}\ b)\ (x,s)
\end{aligned}$$

The remaining cases for E remain unchanged with respect to the calculation in §4.4. The transformations E and B must be introduced via statements. Assignments contain numerical expressions.

$$\begin{aligned}
& \text{assígn}\ (x,e)\ s \\
= {}& x := e\ s \\
= {}& \mathrm{E}\ e\ (x,s) \\
= {}& \text{assìgn}\ (x,\mathrm{E}\ e)\ s
\end{aligned}$$

In conditionals and loops we find boolean expressions.

$$\begin{array}{ll} & \acute{\mathit{cond}}\ (b,s,t)\ u \\ = & b \rightarrow (s\ u) \mathbin{[\!]} (t\ u) \\ = & B\ b\ ((s\ u),(t\ u)) \\ = & \grave{\mathit{cond}}\ (B\ b,s,t)\ u \end{array} \qquad \begin{array}{ll} & \acute{\mathit{while}}\ (b,s)\ t \\ = & \mu(\lambda loop.b \rightarrow (s\ loop) \mathbin{[\!]} t \\ = & \mu(\lambda loop.B\ b\ (s\ loop,t)) \\ = & \grave{\mathit{while}}\ (B\ b,s)\ t \end{array}$$

The fusion law yields the compilation scheme:

$$\begin{array}{rcl}
\mathcal{S}[\![\textbf{skip}]\!]\ s & = & s \\
\mathcal{S}[\![x := E]\!]\ S & = & \mathcal{E}[\![E]\!]\ (x,s) \\
\mathcal{S}[\![S\ ;\ T]\!]\ u & = & \mathcal{S}[\![S]\!]\ (\mathcal{S}[\![T]\!]\ u) \\
\mathcal{S}[\![\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!]\ u & = & \mathcal{B}[\![B]\!]\ (\mathcal{S}[\![S]\!]\ u, \mathcal{S}[\![T]\!]\ u) \\
\mathcal{S}[\![\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}]\!]\ t & = & \mu(\lambda loop.\mathcal{B}[\![B]\!]\ (\mathcal{S}[\![S]\!]\ loop,t)) \\
\\
\mathcal{E}[\![\textbf{var}\ y]\!]\ (x,s) & = & y := \mathrm{LOOKUP}\ x\ s \\
\mathcal{E}[\![\textbf{num}\ n]\!]\ (x,s) & = & x := \mathrm{LITERAL}\ n\ s \\
\mathcal{E}[\![E \oplus F]\!]\ (z,s) & = & \mathcal{E}[\![E]\!]\ (x,(\mathcal{E}[\![F]\!]\ (y,(z := (x \oplus y)\ s)))) \\
\mathcal{E}[\![\textbf{bool}\ B]\!]\ (z,s) & = & \mathcal{B}[\![B]\!]\ (z := \mathrm{TRUE}\ s, z := \mathrm{FALSE}\ s) \\
\\
\mathcal{B}[\![\textbf{var}\ x]\!]\ (s,t) & = & (\mathrm{LOOKUP}\ x) \rightarrow s \mathbin{[\!]} t \\
\mathcal{B}[\![\textbf{true}]\!]\ (s,t) & = & s \\
\mathcal{B}[\![\textbf{false}]\!]\ (s,t) & = & t \\
\mathcal{B}[\![E \ominus F]\!]\ (z,s) & = & \mathcal{E}[\![E]\!]\ (x,(\mathcal{E}[\![F]\!]\ (y,(z := (x \ominus y)\ s)))) \\
\mathcal{B}[\![A\ \textbf{and}\ B]\!]\ (s,t) & = & \mathcal{B}[\![A]\!]\ (\mathcal{B}[\![B]\!]\ (s,t)),t) \\
\mathcal{B}[\![A\ \textbf{or}\ B]\!]\ (s,t) & = & \mathcal{B}[\![A]\!]\ (s,(\mathcal{B}[\![B]\!]\ (s,t)))
\end{array}$$

### 4.5.1 Alternative solutions

Alternative derivations for introducing short circuit evaluation are possible where the syntax need not be changed. But anyhow the evaluation function must 'know' whether the expression has to be translated as an ordinary expression or as a boolean expression.

One solution is to use *mutumorphisms* as described by Fokkinga [35]. A mutumorphism is a set of mutual recursive functions defined on a single data type. The mutumorphism solution would correspond to the situation where $\mathcal{B}[\![\_]\!]$ and $\mathcal{E}[\![\_]\!]$ would be defined as mutual recursive functions on the single type $expression$. Although mutumorphisms posses algebraic properties like *Uniqueness* and *Fusion*, they are quite troublesome due to the tupling involved.

Another solution would be to have an additional inherited attribute indicating the compilation mode, i.e., boolean or arithmetical as done by Aho, Sethi and Ullman [1] §8.4.

It is much more efficient to leave this encoding to the parser that generates the abstract syntax tree.

## 4.6 Adding Procedures: from $\mathcal{W}$ to $\mathcal{C}$

Function procedures are added to $\mathcal{W}$ by extending the syntax with the clauses:

$$\begin{aligned} \mathit{statement} &::= \ \ldots \mid \textbf{return}\ \mathit{expression} \\ \mathit{expression} &::= \ \ldots \mid \textbf{call}\ \mathit{procedure}\ (\mathit{expression}) \\ P \in \mathit{procedure} &::= \ \textbf{proc}\ (\mathit{var})\ \textbf{begin}\ \mathit{statement}\ \textbf{end} \end{aligned}$$

Recursive (function) procedures will be represented by *cyclic* programs. Usually programs are assumed to be finite and recursion is solved semantically by means of a recursive environment mapping procedure names into their denotations. Solving recursion on the syntax level as we do makes calculations a lot simpler, and the syntax as well. Procedures need not have names. Mathematically there is no difference between having the recursion in the programs or in the environment (also see [95] pp 125-126 for a discussion about this). In our framework, abstract syntax described by an algebraic data type defines both finite and infinite programs. Instead of viewing this as a bug, we regard it as a feature. Pragmatically it makes a world of difference not to take advantage of syntactic recursion, one should consult Meyer [73] for examples where the use of environments makes derivations less elegant.

Using cyclic programs it is impossible to describe dynamic binding. Syntactic recursion is static binding at its extreme.

As a matter of fact, the introduction of cycles in the semantics via **while**-loops is already a major source of disruption in our calculations since we must take resort to fixed point induction every time we are proving promotability of improving transformations. For cyclic programs this is done once and for all in the proof of the fusion law.

**example** An example recursive procedure is $nfib$

```
μ(λnfib  .  proc (n)
            begin
              if (n ≤ 1) then return 1
                          else return (1 + nfib (n − 1) + nfib (n − 2))
              fi
            end)
```

### 4.6.1 Dynamic semantics of functions

Leaving the definition of a modified static semantics as future work, we now turn our attention towards a dynamic semantics for $\mathcal{C}$. Statement continuation will be of type

$env \rightarrow \mathbb{N}$ instead of $env \rightarrow env$ because function procedures return a value.

$$
\begin{aligned}
\text{RETURN}\ x\ s\ \eta &= \eta\ x \\
x := \text{CALL}\ ((p,y),z)\ s\ \eta &= s\ \eta[x := v]\ \mathit{where}\ v = p\ (\eta_0[y := \eta\ z]) \\
\text{EXIT}\ s\ \eta &= \bot
\end{aligned}
$$

The CALL-instruction $x := \text{CALL}\ ((p,y),z)$ calls procedure $p$ with an initial environment in which the formal argument $y$ is bound to the value of the actual argument $z$. The value computed by $p$ is assigned to $x$.

The extra valuation functions for the new syntactic elements are defined using the new semantic operations. The meaning of complete programs must be redefined as well.

$$
\begin{aligned}
\mathcal{M}[\![\textbf{prog}\ S]\!] &= \mathcal{S}[\![S]\!]\ (\text{EXIT}\ \bot)\ \eta_0 \\
\mathcal{S}[\![\textbf{return}\ E]\!]\ s &= \mathcal{E}[\![E]\!]\ (x, \text{RETURN}\ x\ s) \\
\mathcal{E}[\![\textbf{call}\ P(E)]\!]\ (x,s) &= \mathcal{E}[\![E]\!]\ (y, x := \text{CALL}\ (\mathcal{P}[\![P]\!], y)\ s) \\
\mathcal{P}[\![\_]\!] &\in procedure \rightarrow stat \| var \\
\mathcal{P}[\![\textbf{proc}\ (x)\ \textbf{begin}\ S\ \textbf{end}]\!] &= (\mathcal{S}[\![S]\!]\ (\text{EXIT}\ \bot), x)
\end{aligned}
$$

A procedure or program that does not RETURN explicitly, implicitly returns $\bot$.

### 4.6.2 Introducing the dump

The semantics of a procedure call

$$x := \text{CALL}\ ((p,y),z)\ s\ \eta \quad = \quad s\ \eta[x := v]\ \mathit{where}\ v = p\ (\eta_0[y := \eta\ z])$$

does not reflect the standard subroutine call, where evaluation on the caller's side is temporarily suspended and control is transferred from caller to callee which eventually returns its result to back to the caller. This implicit evaluation order will be explicated by introducing yet another continuation, the *dump*, which represents the suspended computation of the caller of the currently executing procedure. The dump continuation has type $\mathbb{N} \rightarrow \mathbb{N}$, and should be strict. Given the result of the callee, the caller may resume computing its result, but if the callee evaluates to $\bot$ the whole computation has to fail.

The injective function $C \in (env \rightarrow \mathbb{N}) \rightarrow (env \rightarrow dump \rightarrow \mathbb{N})$ maps an abstract continuation $\acute{s}$ that does not expect a dump, into a concrete one $\grave{s} = C\ \acute{s}$ that does expect a dump.

$$C\ \acute{s}\ \eta\ \delta \quad = \quad \delta\ (\acute{s}\ \eta)$$

Its left-inverse $A \in (env \rightarrow dump \rightarrow \mathbb{N}) \rightarrow (env \rightarrow \mathbb{N})$ maps a concrete continuation $\grave{s} = C\ \acute{s}$ back into an abstract one $\acute{s} = A\ \grave{s}$.

$$
\begin{aligned}
& A\ \grave{s}\ \eta \\
= \quad & \grave{s} = C\ \acute{s}
\end{aligned}
$$

$$\begin{array}{ll}
 & A\ (C\ \acute{s})\ \eta \\
= & \quad \text{wish} \\
 & \acute{s}\ \eta \\
= & \quad \text{aiming at folding C} \\
 & id\ (\acute{s}\ \eta) \\
= & \quad \text{fold} \\
 & C\ \acute{s}\ \eta\ id \\
= & \quad C\ \acute{s} = \grave{s} \\
 & \grave{s}\ \eta\ id
\end{array}$$

Thus $A\ \grave{s}\ \eta = \grave{s}\ \eta\ id$ is a left-inverse of $C$. The new semantics is calculated using the fact that $A \circ C = id \in env \to \mathbb{N}$.

$$\begin{array}{ll}
 & \acute{\mathcal{M}}[\![\textbf{prog}\ S]\!] \\
= & \quad \text{wish} \\
 & \grave{\mathcal{M}}[\![\textbf{prog}\ S]\!] \\
= & \quad \text{unfold} \\
 & \acute{\mathcal{S}}[\![S]\!]\ (\text{EXIT}\ \bot)\ \eta_0 \\
= & \quad A \circ C = id \\
 & (A \circ C \circ \acute{\mathcal{S}}[\![S]\!])\ (\text{EXIT}\ \bot)\ \eta_0 \\
= & \quad \text{assume } C \circ \acute{\mathcal{S}}[\![S]\!] = \grave{\mathcal{S}}[\![S]\!] \circ C \\
 & (A \circ \acute{\mathcal{S}}[\![S]\!] \circ C)\ (\text{EXIT}\ \bot)\ \eta_0 \\
= & \quad \text{unfold } A \\
 & (\acute{\mathcal{S}}[\![S]\!] \circ C)\ (\text{EXIT}\ \bot)\ \eta_0\ id
\end{array}$$

The new semantics will be derived by proving the assumption $C \circ \acute{\mathcal{S}}[\![S]\!] = \grave{\mathcal{S}}[\![S]\!] \circ C$ using the induction principle for catamorphisms.

$$\begin{array}{rcl}
C \circ \acute{\mathcal{S}}[\![S]\!] = \grave{\mathcal{S}}[\![S]\!] \circ C & & \\
\wedge\ C \circ \acute{\mathcal{E}}[\![E]\!] = \grave{\mathcal{E}}[\![E]\!] \circ id\|C & & \\
\wedge\ (C\|id)\ \acute{\mathcal{P}}[\![P]\!] = \grave{\mathcal{P}}[\![P]\!] & \Leftarrow & (C \circ sk\acute{i}p = sk\grave{i}p \circ C)\ \wedge \\
 & & (C \circ s\acute{e}q\ (\acute{s}, \acute{t}) = s\grave{e}q\ (\grave{s}, \grave{t}) \circ C \\
 & & \quad \Leftarrow C \circ \acute{s} = \grave{s} \circ C\ \wedge\ C \circ \acute{t} = \grave{t} \circ C)\ \wedge \\
 & & (C \circ ass\acute{i}gn\ (x, \acute{e}) = ass\grave{i}gn\ (x, \grave{e}) \circ C \\
 & & \quad \Leftarrow C \circ \acute{e} = \grave{e} \circ id\|C)\ \wedge \\
 & & (C \circ c\acute{o}nd\ (\acute{b}, \acute{s}, \acute{t}) = c\grave{o}nd\ (\grave{b}, \grave{s}, \grave{t}) \circ C \\
 & & \quad \Leftarrow C \circ \acute{b} = \grave{b} \circ id\|C\ \wedge \\
 & & \quad\quad C \circ \acute{s} = \grave{s} \circ C\ \wedge\ C \circ \acute{t} = \grave{t} \circ C)\ \wedge \\
 & & (C \circ wh\acute{i}le\ (\acute{b}, \acute{s}) = wh\grave{i}le\ (\grave{b}, \grave{s}) \circ C
\end{array}$$

$$
\begin{aligned}
&\quad \Leftarrow C \circ \acute{b} = \grave{b} \circ id \| C \ \wedge\ C \circ \acute{s} = \grave{s} \circ C \ \wedge) \\
&(C \circ o\acute{p}er\ (\acute{e}, \acute{f}) = o\grave{p}er\ (\grave{e}, \grave{f}) \circ id \| C \\
&\quad \Leftarrow C \circ \acute{e} = \grave{e} \circ id \| C \ \wedge\ C \circ \acute{f} = \grave{f} \circ id \| C) \ \wedge \\
&(C \circ c\acute{a}ll\ (\acute{p}, \acute{e}) = c\grave{a}ll\ (\grave{p}, \grave{e}) \circ id \| C \\
&\quad \Leftarrow C \circ \acute{e} = \grave{e} \circ id \| C \ \wedge\ (C \| id)\ \acute{p} = \grave{p}) \ \wedge \\
&((C \| id)\ pr\acute{o}c\ (\acute{s}, x) = pr\grave{o}c\ (\grave{s}, x) \\
&\quad \Leftarrow C \circ \acute{s} = \grave{s} \circ C)
\end{aligned}
$$

Writing the above theorem with bound variables would make it even more incomprehensible.

**strictness** Now that programs may be cyclic, we must check that C is strict. This is obvious, as we required dump continuations to be strict. The meaning of **skip** statements and sequencing remains unchanged.

$$
\begin{aligned}
& C\ (skip\ s) \\
= &\quad \text{unfold} \\
& C\ s \\
= &\quad \text{extract} \\
& skip\ (C\ s)
\end{aligned}
$$

$$
\begin{aligned}
& C\ ((seq\ (\acute{s}, \acute{t})\ u) \\
= &\quad \text{unfold} \\
& C\ (\acute{s}\ (\acute{t}\ u)) \\
= &\quad \text{IH twice} \\
& \grave{s}\ (\grave{t}\ (C\ u)) \\
= &\quad \text{extract} \\
& seq\ (\grave{s}, \grave{t})\ (C\ u)
\end{aligned}
$$

If we use the perhaps more obvious statement $C\ (\acute{\mathcal{S}}[\![\textbf{skip}]\!]\ s) = \grave{\mathcal{S}}[\![\textbf{skip}]\!]\ s$ the result is an unsatisfying, but correct, semantics.

$$
\begin{aligned}
& C\ (\acute{\mathcal{S}}[\![\textbf{skip}]\!]\ s)\ \eta\ \delta \\
= &\quad \text{unfold} \\
& C\ s\ \eta\ \delta \\
= &\quad \text{unfold} \\
& \delta\ (s\ \eta) \\
= &\quad \text{synthesize} \\
& \mathrm{SKIP}\ s\ \eta\ \delta \\
= &\quad \text{extract} \\
& \grave{\mathcal{S}}[\![\textbf{skip}]\!]\ s\ \eta\ \delta
\end{aligned}
$$

It forces the introduction of a weird run-time instruction SKIP that returns immediately.

**assignment** Assignments pose no problems.

$$
C\ (ass\acute{i}gn\ (x, \acute{e})\ s)
$$

$=$ unfold
$C\ (\acute{e}\ (x, s))$
$=$ IH
$\grave{e}\ (x, C\ s)$
$=$ extract
$\mathit{ass\grave{\imath}gn}\ (x, \grave{e})\ (C\ s)$

**return** The side effect of applying C to RETURN-statements is a new instruction.

$C\ (\mathit{ret\acute{u}rn}\ \acute{e}\ s)$
$=$ unfold
$C\ (\acute{e}\ (x, \mathrm{RET\acute{U}RN}\ x\ s))$
$=$ IH
$\grave{e}\ (x, C\ (\mathrm{RET\acute{U}RN}\ x\ s))$
$=$ abutting calculation
$\grave{e}\ (x, \mathrm{RET\grave{U}RN}\ x\ (C\ s))$
$=$ extract
$\mathit{ret\grave{u}rn}\ \grave{e}\ (C\ s)$

$C\ (\mathrm{RET\acute{U}RN}\ x)\ s\ \eta\ \delta$
$=$ unfold
$\delta\ (\mathrm{RET\acute{U}RN}\ x\ s\ \eta)$
$=$ evaluate
$\delta\ (\eta\ x)$
$=$ synthesize
$\mathrm{RET\grave{U}RN}\ (C\ s)\ \eta\ \delta$

The instruction $\mathrm{RET\grave{U}RN}\ s\ \eta\ \delta = \delta\ (\eta\ x)$ captures the intuition of the statement **return** E, namely return the value of E to the caller of the current function.

**conditional** The conditional statement takes five.

$C\ (\mathit{cond}\ (\acute{b}, \acute{s}, \acute{t}))\ u$
$=$ unfold
$C\ (\acute{b}\ (x, \mathrm{LOOKUP}\ x \rightarrow (\acute{s}\ u)\ []\ (\acute{t}\ u)))$
$=$ IH
$\grave{b}\ (x, C\ (\mathrm{LOOKUP}\ x \rightarrow (\acute{s}\ u)\ []\ (\acute{t}\ u)))$
$=$ property conditional
$\grave{b}\ (x, \mathrm{LOOKUP}\ x \rightarrow (C\ (\acute{s}\ u))\ []\ (C\ (\acute{t}\ u))))$
$=$ IH twice
$\grave{b}\ (x, \mathrm{LOOKUP}\ x \rightarrow (\grave{s}\ (C\ u))\ []\ (\grave{t}\ (C\ u))))$
$=$ extract
$\mathit{cond}\ (\grave{b}, \grave{s}, \grave{t})\ (C\ u)$

**while-loops** For **while**-loops the following lemma is needed.

$$g\ \mu(\lambda x.A[x]) = \mu(\lambda x.B[x]) \quad \Leftarrow \quad g\ A[x] = B[g\ x] \tag{4.3}$$

which follows directly from the fixed point fusion theorem and the two λ-fusion laws.

$$
\begin{array}{ll}
 & \mathsf{C}\ (\acute{while}\ (\acute{b}, \acute{s})\ t) \\
= & \quad \text{unfold} \\
 & \mathsf{C}\ (\mu(\lambda loop.\acute{b}\ (x, \mathrm{LOOKUP}\ x \to (\acute{s}\ loop) \mathbin{[\!]} t))) \\
= & \quad \text{lemma, previous calculations for conditional} \\
 & \mu(\lambda loop.\grave{b}\ (x, \mathrm{LOOKUP}\ x \to (\grave{s}\ loop) \mathbin{[\!]} (\mathsf{C}\ t))) \\
= & \quad \text{extract} \\
 & \grave{while}\ (\grave{b}, \grave{s})\ (\mathsf{C}\ t)
\end{array}
$$

**expressions** Having shown that C promotes over all operations of $\mathcal{S}[\![\_]\!]$, we now must show that C also promotes over the operations of $\mathcal{E}[\![\_]\!]$. This determines a new run-time operation.

$$
\begin{array}{ll}
 & \mathsf{C}\ (\acute{oper}\ (\acute{a}, \acute{b})\ (x, s)) \\
= & \quad \text{unfold} \\
 & \mathsf{C}\ (\acute{a}\ (y, \acute{b}\ (z, x :\acute{=} (y \textcircled{2} z)\ s))) \\
= & \quad \text{IH twice} \\
 & \grave{a}\ (y, \grave{b}\ (z, \mathsf{C}\ (x :\acute{=} (y \textcircled{2} z)\ s))) \\
= & \quad \text{abutting calculation} \\
 & \grave{a}\ (y, \grave{b}\ (z, x :\grave{=} (y \textcircled{2} z)\ (\mathsf{C}\ s)))) \\
= & \quad \text{extract} \\
 & \grave{oper}\ (\grave{a}, \grave{b})\ (x, \mathsf{C}\ s)
\end{array}
\qquad
\begin{array}{ll}
 & \mathsf{C}\ (x :\acute{=} (y \textcircled{2} z))\ s\ \eta\ \delta \\
= & \quad \text{unfold} \\
 & \delta\ (x :\acute{=} (y \textcircled{2} z)\ s\ \eta) \\
= & \quad \text{evaluate} \\
 & \delta\ (s\ \eta[x := \eta\ y \textcircled{2} \eta\ z]) \\
= & \quad \text{fold} \\
 & \mathsf{C}\ s\ \eta[x := \eta\ y \textcircled{2} \eta\ z]\ \delta \\
= & \quad \text{synthesize} \\
 & x :\grave{=} (y \textcircled{2} z)\ (\mathsf{C}\ s)\ \eta\ \delta
\end{array}
$$

For expressions compiled into LITERAL and LOOKUP we find in a similar fashion

$$
\begin{array}{rcl}
(x := \mathrm{LOOKUP}\ y)\ s\ \eta\ \delta & = & s\ \eta[x := \eta\ y]\ \delta \\
(x := \mathrm{LITERAL}\, n)\ s\ \eta\ \delta & = & s\ \eta[x := n]\ \delta
\end{array}
$$

**procedure call** The reason why doing the current derivation is to implement function calls.

$$
\begin{array}{ll}
 & \mathsf{C}\ (\acute{call}\ ((\acute{p}, y), \acute{e})\ (x, s)) \\
= & \quad \text{unfold} \\
 & \mathsf{C}\ (\acute{e}\ (z, x :\acute{=} \mathrm{CALL}\ ((\acute{p}, y), z)\ s)) \\
= & \quad \text{IH} \\
 & \grave{e}\ (z, \mathsf{C}\ (x :\acute{=} \mathrm{CALL}\ ((\acute{p}, y), z)\ s)) \\
= & \quad \text{synthesize, see below}
\end{array}
$$

$$\begin{array}{ll}
 & \grave{e}\ (z, x \mathrel{:\grave{=}} \mathrm{CALL}\ (((C\|id)\ \acute{p}, y), z)\ (C\ s))) \\
= & \quad \text{IH} \\
 & \grave{e}\ (z, x \mathrel{:\grave{=}} \mathrm{CALL}\ ((\grave{p}, y), z)\ (C\ s))) \\
= & \quad \text{extract} \\
 & c\grave{a}ll\ ((\grave{p}, y), \grave{e})\ s
\end{array}$$

In the fourth step of this calculation the assumption has been made that $C\ (x \mathrel{:\acute{=}} \mathrm{CALL}\ ((\acute{p}, y), z))\ s = x \mathrel{:\grave{=}} \mathrm{CALL}\ (((C\|id)\ \acute{p}, y), z)\ (C\ s)$ which remains to be shown.

$$\begin{array}{ll}
 & C\ (x \mathrel{:\acute{=}} \mathrm{CALL}\ ((p, y), z))\ s\ \eta\ \delta \\
= & \quad \text{unfold} \\
 & \delta\ (x \mathrel{:\acute{=}} \mathrm{CALL}\ ((p, y), z)\ s\ \eta) \\
= & \quad \text{evaluate} \\
 & (\delta \circ (\lambda v.s\ \eta[x := v]))\ (p\ (\eta_0[y := \eta\ z])) \\
= & \quad \text{law for } \lambda\text{, note that the dump remains strict} \\
 & (\lambda v.\delta\ (s\ \eta[x := v]))\ (p\ (\eta_0[y := \eta\ z])) \\
= & \quad \text{fold} \\
 & C\ p\ (\eta_0[y := \eta\ z])\ (\lambda v.C\ s\ \eta[x := v]\ \delta)) \\
= & \quad \text{synthesize} \\
 & x \mathrel{:\grave{=}} \mathrm{CALL}\ ((C\ p, y), z)\ (C\ s)\ \eta\ \delta
\end{array}$$

**Function entrance**

The last remaining case is $(C\|id) \circ pr\acute{o}c$.

$$\begin{array}{ll}
 & (C\|id)\ (pr\acute{o}c\ (\acute{s}, x)) \\
= & \quad \text{unfold} \\
 & (C\|id)\ (\acute{s}\ (\mathrm{EX\acute{I}T}\ \bot), x) \\
= & \quad \text{IH} \\
 & (\grave{s}\ (C\ (\mathrm{EX\acute{I}T}\ \bot)), x) \\
= & \quad \text{abutting calculation} \\
 & (\grave{s}\ (\mathrm{EX\grave{I}T}\ \bot), x) \\
= & \quad \text{extract} \\
 & pr\grave{o}c\ (\grave{s}, x)
\end{array}$$

$$\begin{array}{ll}
 & C\ (\mathrm{EX\acute{I}T}\ s)\ \eta\ \delta \\
= & \quad \text{unfold} \\
 & \delta\ (\mathrm{EX\grave{I}T}\ s\ \eta) \\
= & \quad \text{evaluate} \\
 & \delta\ \bot \\
= & \quad \text{synthesize} \\
 & \mathrm{EX\grave{I}T}\ s\ \eta\ \delta
\end{array}$$

### 4.6.3 Final Semantics

Applying the catamorphism induction rule yields the final semantics for $\mathcal{C}$.

$$\mathcal{M}[\![\textbf{prog}\ P]\!] = \mathcal{S}[\![P]\!]\ (\mathrm{EXIT}\ \bot)\ \eta_0\ \mathrm{id}$$

$$\begin{array}{rcl}
\mathcal{S}[\![\textbf{skip}]\!]\ s &=& s \\
\mathcal{S}[\![x := E]\!]\ s &=& \mathcal{E}[\![E]\!]\ (x, s) \\
\mathcal{S}[\![S\ ;\ T]\!]\ u &=& \mathcal{S}[\![S]\!]\ (\mathcal{S}[\![T]\!]\ u) \\
\mathcal{S}[\![\textbf{return}\ E]\!]\ s &=& \mathcal{E}[\![E]\!]\ (x, \mathrm{RETURN}\ x\ s) \\
\mathcal{S}[\![\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}]\!]\ u &=& \mathcal{E}[\![B]\!]\ (x, \mathrm{LOOKUP}\ x \to \mathcal{S}[\![S]\!]\ u\ [\!]\ \mathcal{S}[\![T]\!]\ u) \\
\mathcal{S}[\![\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}]\!]\ t &=& \mu(\lambda loop. \mathcal{E}[\![B]\!]\ (x, \mathrm{LOOKUP}\ x \to (\mathcal{S}[\![S]\!]\ loop)\ [\!]\ t))
\end{array}$$

$$\begin{array}{rcl}
\mathcal{E}[\![\textbf{var}\ y]\!]\ (x, s) &=& x := \mathrm{LOOKUP}\ y\ s \\
\mathcal{E}[\![\textbf{num}\ n]\!]\ (x, s) &=& x := \mathrm{LITERAL}\ n\ s \\
\mathcal{E}[\![\textbf{true}]\!]\ (x, s) &=& x := \mathrm{TRUE}\ s \\
\mathcal{E}[\![\textbf{false}]\!]\ (x, s) &=& x := \mathrm{FALSE}\ s \\
\mathcal{E}[\![E\ ②\ F]\!]\ (x, s) &=& \mathcal{E}[\![E]\!]\ (y, \mathcal{E}[\![F]\!]\ (z, x := (y\ ②\ z)\ s)) \\
\mathcal{E}[\![\textbf{call}\ P(E)]\!]\ (x, s) &=& \mathcal{E}[\![E]\!]\ (y, x := \mathrm{CALL}\ (\mathcal{P}[\![P]\!], y)\ s) \\
\mathcal{P}[\![\textbf{proc}\ (x)\ \textbf{begin}\ s\ \textbf{end}]\!] &=& (\mathcal{S}[\![s]\!]\ (\mathrm{EXIT}\ \bot), x)
\end{array}$$

The run-time operations indeed are very close to concrete machine instructions

$$\begin{array}{rcl}
\mathrm{EXIT}\ s\ \eta\ \delta &=& \delta\ \bot \\
\mathrm{RETURN}\ x\ s\ \eta\ \delta &=& \delta\ (\eta\ x) \\
(x := \mathrm{LOOKUP}\ y)\ s\ \eta\ \delta &=& s\ \eta[x := \eta\ y]\ \delta \\
(x := \mathrm{LITERAL}\ n)\ s\ \eta\ \delta &=& s\ \eta[x := n]\ \delta \\
x := \mathrm{CALL}\ ((p, y), z)\ s\ \eta\ \delta &=& p\ (\eta_0[y := \eta\ z])\ (\lambda v.s\ \eta[x := v]\ \delta) \\
x := (y\ ②\ z)\ s\ \eta\ \delta &=& s\ \eta[x := \eta\ y\ ②\ \eta\ z]\ \delta)
\end{array}$$

## 4.7 Tail call elimination

Suppose the context condition holds that any variable appearing in a program is defined (occurs on the lhs of an assignment) before it is used (occurs on the rhs of an assignment), then the CALL instruction can be refined as

$$x := \mathrm{CALL}\ ((p, y), z)\ s\ \eta\ \delta \quad = \quad p\ (\eta[y := \eta\ z])\ (\lambda v.s\ \eta[x := v]\ \delta)$$

Thus the current environment $\eta$ may be passed to $p$ instead of the empty environment $\eta_0$. This modified CALL instruction allows certain recursive calls can be replaced by iteration. When a procedure returns by calling a procedure (either itself or another), that call is said to be a *tail call*. We can replace such a call by a jump, this not only saves time, but also space.

$$
\begin{array}{ll}
 & (x := \mathrm{CALL}\ ((p, y), z) \circ \mathrm{RETURN}\ x)\ s\ \eta\ \delta \\
= & \quad \text{evaluate CALL} \\
 & p\ (\eta[y := \eta\ z])\ (\lambda v.\mathrm{RETURN}\ x\ s\ \eta[x := v]\ \delta) \\
= & \quad \text{evaluate RETURN} \\
 & p\ (\eta[y := \eta\ z])\ \delta \\
= & \quad \text{devaluate LOOKUP and GOTO} \\
 & (y := \mathrm{LOOKUP}\ z \circ \mathrm{GOTO}\ p)\ s\ \eta\ \delta
\end{array}
$$

Incorporating this into our translation scheme gives

$$\mathcal{S}[\![\textbf{return}\ (\textbf{call}\ \mathrm{P(E)})]\!]\ s \quad = \quad \mathcal{E}[\![\mathrm{E}]\!]\ (x, \mathrm{GOTO}\ p\ s)\ \mathit{where}\ (p, x) = \mathcal{P}[\![\mathrm{P}]\!]$$

Note that this definition is not homomorphic, but can be made so easily.

## 4.8 Concrete Implementation

The run-time operations are now so low-level that they can be implemented directly in some assembler language. The CALL and RETURN instruction would require a little more work; the dump must be defunctionalized into a stack of environment and return address pairs. If C is used as a high-level assembler language that provides simple function calls[1], our example function is translated into:

```
int nfib (int n){
   register int r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14;
   r0 = 1; r1 = n;
   r2 = (r1 <= r0);
   if (r2) {goto l}
   r3 = 1; r4 = n; r5 = r4-r3; r6 = nfib(r5);
   r7 = n; r8 = 2; r9 = r7-r8; r10 = nfib(r9);
   r11 = 1; r12 = r11+r10; r13 = r12+r6;
   return(r13);
l: r14 = 1;
   return(r14);
   }
```

When compiled using `gcc -O` on a *Data General AV300* this gives an nfib number of 1.200.000. Formally derived compilers can be efficient!

[1]unfortunately most C compilers do not treat tail calls properly

# Chapter 5

# A Taxonomy of Backtracking

*Can't you find another way of doing it baby*
*Can't you*

Sam and Dave

An increasing number of programming languages provide *backtracking* as a form of control structure: Snobol [46], Icon [45], Prolog [20] and Summer [54], to mention just a few. Not surprisingly, much research goes on in this area, both practically as well as theoretically. There seems to be, however, a great lull in the conversation between these two worlds. On the one hand, practical implementations are presented in an ad hoc fashion with very little motivation, whereas denotational and operational semantics of backtracking usually stay far away from any implementation on some concrete machine. Hence the relationship between formal semantics and concrete implementations usually stays unclear.

This chapter derives several implementations for a simple backtrack language $\mathcal{B}$. This language is what remains from a logic language like Prolog when all syntactic structure in literals is abstracted away and semantic concepts such as unification and substitution are ignored; $\mathcal{B}$= *logic programming - logic*. Mainly for historical reasons, we will interpret elementary $\mathcal{B}$ actions as terminal symbols, hence the interpretation of a $\mathcal{B}$ script as a (possibly ambiguous) context free grammar.

The initial meaning function will map a script onto a relation that coincides with a Definite Clause Grammar [42] as known from logic programming. These relations are subsequently implemented as functions. In order to capture a *backtracking* implementation of nondeterministic choice, list-valued rather than set-valued functions are used. Depending on the dual choice of specifying elementary actions as relations or as functions we get a spectrum of direct semantics for $\mathcal{B}$, with on the two extremes respectively those of de Vink [30] and of Salter and Jones [94]. We then continue by transforming the former semantics until we arrive at a number of low level implementations of backtracking among which the backtracking part of the WAM [110] and the Recursive Backup Machine [57].

The use of continuations for describing (the semantics of) backtracking has now become quite common [24, 57, 62, 31, 26, 103, 25], but deriving this from a relational specification and further transformation into concrete implementations was not done before. To our best knowledge we have presented the first systematic derivation of the backtracking part of the WAM, a derivation of the unification part is given by Kursawe [59].

## 5.1 A calculus of relations

The calculus of relations that we use is based on [29]. Given a (typed) binary relation $R \subseteq B \| A$ we write $R \in B \leftrightarrow A$ as $b\ R\ a$ or $R(b, a)$ if $(b, a) \in R$. We assume that the arguments of our relations are flat domains; for flat domains $D$ the powerdomain consists of precisely the non-$\bot$ subsets of $D$.

Relational composition is defined in analogy with function composition as:

$$\begin{array}{rcl} \circ & \in & (C \leftrightarrow B \| B \leftrightarrow A) \to (C \leftrightarrow A) \\ R \circ S & = & \{(c, a) \mid (\exists b \in B :: c\ R\ b \wedge b\ S\ a)\} \end{array}$$

Sometimes we will use ; as notation for $\tilde{\circ}$. The union of two relations is defined as:

$$\begin{array}{rcl} \cup & \in & (B \leftrightarrow A \| B \leftrightarrow A) \to (B \leftrightarrow A) \\ R \cup S & = & \{(a, b) \mid a\ R\ b \vee a\ S\ b\} \end{array}$$

A function $f \in B \to A$ is mapped into the relation $F \in B \leftrightarrow A$ by taking its *graph*:

$$\begin{array}{rcl} |\_| & \in & (A \to B) \to (B \leftrightarrow A) \\ |f| & = & \{(b, a) \mid b = f\ a\} \end{array}$$

A set-valued function $f \in A \to \{B\}$ is mapped into $F \in B \leftrightarrow A$ by taking its *choice*:

$$\begin{array}{rcl} \mathrm{Ch} & \in & (A \to \{B\}) \to (B \leftrightarrow A) \\ \mathrm{Ch}\ f & = & \{(b, a) \mid b \in f\ a\}. \end{array}$$

A relation $F \in B \leftrightarrow A$ is mapped into the function $f \in A \to \{B\}$ by taking its *breadth*:

$$\begin{array}{rcl} \mathrm{Br} & \in & (B \leftrightarrow A) \to (A \to \{B\}) \\ \mathrm{Br}\ F\ a & = & \{b \in B \mid b\ F\ a\} \end{array}$$

$\mathrm{Br}$ and $\mathrm{Ch}$ establish a bijection between relations and set valued functions.

$$f = \mathrm{Br}\ F \quad \equiv \quad F = \mathrm{Ch}\ f$$

In the sequel we use that:

$$\begin{array}{ll} & (\mathrm{Br} \circ \mathrm{Ch})\ f\ a \\ = & \mathrm{Br}\ (\mathrm{Ch}\ f)\ a \\ = & \{b \in B \mid b\ \mathrm{Ch}\ f\ a\} \\ = & \{b \in B \mid b \in f\ a\} \\ = & f\ a \end{array}$$

### 5.1.1 $\mathrm{Br}$ promotion laws

The next breadth promotion laws will be used in the next section to map relations into functions.

$$\begin{aligned}
\mathrm{Br} \circ (\circ) &= \circledcirc \circ \mathrm{Br} \| \mathrm{Br} \\
\mathrm{Br} \circ \cup &= \hat{\cup} \circ \mathrm{Br} \| \mathrm{Br} \\
\mathrm{Br} \circ |\_| &= (\{\_\} \circ) \circ \mathrm{id} \\
\mathrm{Br} \circ \mathrm{Ch} &= \mathrm{id}
\end{aligned}$$

where $f \circledcirc g = \cup/ \circ f* \circ g$. Kleiski composition of set-valued functions $f \circledcirc g$ first applies $g$ to its argument, which yields a set of answers. Then $f$ is applied to each of the elements of this set, giving a set of set of answers which is flattened ($\cup/$) into a single set in the end.

The next corollaries to the above laws may be used as peephole optimizations.

$$\begin{aligned}
\mathrm{Br}\ (|f| \circ |g|) &= \mathrm{Br}\ |f \circ g| \\
&= \{\_\} \circ f \circ g \\
\mathrm{Br}\ (|f| \circ g) &= f* \circ \mathrm{Br}\ g \\
\mathrm{Br}\ (f \circ |g|) &= \mathrm{Br}\ f \circ g
\end{aligned}$$

$\mathrm{Br}$ is a homomorphism on relations built from $\circ, \cup, |\_|$ and $\mathrm{Ch}$.

## 5.2 The Language $\mathcal{B}$

The abstract syntax of $\mathcal{B}$ is given by the grammar:

$$\begin{aligned}
\mathit{script} &::= \mathit{expr} \\
E, F, G \in \mathit{expr} &::= \textbf{empty} \mid \textbf{fail} \mid \mathit{symbol}'' \mid \mathit{expr}\ \textbf{;}\ \mathit{expr} \mid \mathit{expr}\textbf{,}\mathit{expr}
\end{aligned}$$

Throughout the sequel C varies over $\mathit{symbol}$. To save parenthesis we have the convention that **,** binds stronger than **;**. The choice of symbols, **,** for sequential composition and **;** for alternative composition follows usual Prolog convention. In Chapter 4 semicolon **;** was used for sequential composition. Don't get them mixed up.

Informally a symbol C" denotes the requirement of recognizing at the head of the input that very symbol, or fail otherwise; this may be seen as an abstraction of a unification action in full Prolog. The constant **fail** denotes immediate failure, the constant **empty** denotes instant success; juxtaposition $E$**,**$F$ denotes the requirement of first recognizing $E$ and then $F$; alternation $E$ **;** $F$ denotes the requirement of recognizing either $E$ or $F$.

## 5.3 Relational Semantics

The semantic function $\mathcal{M}[\![\_]\!] \in script \to recognizer$, gives the meaning of a script. The function $\mathcal{E}[\![\_]\!] \in expr \to nonterminal$ gives the meaning of an expression. The semantic domains are defined as:

$$\begin{array}{rcl} R, S \in nonterminal & = & input \leftrightarrow input \\ recognizer & = & \mathbb{B} \leftrightarrow input \end{array}$$

The evaluation functions that map expressions and scripts into their denotations are defined by means of a catamorphism.

$$\begin{array}{rcl} \mathcal{M}[\![\textbf{script}\ E]\!] & = & EOS \circ \mathcal{E}[\![E]\!] \\ \mathcal{E}[\![\textbf{fail}]\!] & = & FAIL \\ \mathcal{E}[\![\textbf{empty}]\!] & = & EMPTY \\ \mathcal{E}[\![C"]\!] & = & READ\ C \\ \mathcal{E}[\![E,F]\!] & = & \mathcal{E}[\![F]\!] \circ \mathcal{E}[\![E]\!] \\ \mathcal{E}[\![E\ ;\ F]\!] & = & \mathcal{E}[\![E]\!] \cup \mathcal{E}[\![F]\!] \end{array}$$

To complete our semantics, we must provide definitions for the elementary predicates EMPTY, FAIL, READ C and the end-of-sentence test EOS.

There are essentially two ways of defining these primitive relations; as the $|\_|$ of a non set-valued function, or equivalently directly as a relation or as the choice of a set-valued function. In order to capture the notion of *backtracking* however, we use *lists* instead of *sets* (see also [40]).

Defining primitive relations as Definite Clause Grammars, yields after Br-promotion the direct semantics for $\mathcal{B}$ as given by de Vink [30]. The semantics defined using $|\_|$ coincides with that of Salter and Jones [94]. Thus we have, in a uniform way, derived two seemingly unrelated direct semantics for backtracking.

### 5.3.1 Direct semantics à la de Vink

The direct semantics of [30] results from defining the primitive relations as *Definite Clause Grammars* that are commonly used in Prolog.

$$\begin{array}{rcl} EMPTY & = & \{(j,i) \mid j = i\} \\ FAIL & = & \{\} \\ READ\ C & = & \{(j,i) \mid i = C \rightarrowtail\!\!\!+ j\} \\ EOS & = & \{(i = [\,], i)\} \end{array}$$

If we are not interested in the number of unsuccessful parses, we may define $EOS = \{(true, [\,])\}$.

Taking the breadth of these elementary relations gives the following set of list-valued functions:

$$\begin{aligned}
\mathrm{FAIL}\ i &= [\,] \\
\mathrm{EMPTY}\ i &= [i] \\
\mathrm{READ}\ C\ (C' \rightarrowtail\!\!+ i) &= (C = C') \rightarrow [i] \mathbin{[\!]} [\,] \\
(f \gg g)\ i &= f\ i \mathbin{+\!\!+} g\ i \\
f \circledcirc g &= \mathbin{+\!\!+}/ \circ f* \circ g \\
\mathrm{EOS}\ i &= (i = [\,]) \rightarrow [\mathit{true}] \mathbin{[\!]} [\,]
\end{aligned}$$

and valuation functions for the semantics of de Vink are

$$\begin{aligned}
\mathcal{M}[\![\_]\!] &\in \mathit{script} \rightarrow (\mathit{input} \rightarrow \mathbb{B}*) \\
\mathcal{M}[\![\textbf{script}\ E]\!]\ i &= (\mathrm{EOS} \circledcirc \mathcal{E}[\![E]\!])\ i \\
\mathcal{E}[\![\_]\!] &\in \mathit{expr} \rightarrow (\mathit{input} \rightarrow \mathit{input}*) \\
\mathcal{E}[\![\textbf{fail}]\!] &= \mathrm{FAIL} \\
\mathcal{E}[\![\textbf{empty}]\!] &= \mathrm{EMPTY} \\
\mathcal{E}[\![C"]\!] &= \mathrm{READ}\ C \\
\mathcal{E}[\![E\ ;\ F]\!] &= \mathcal{E}[\![E]\!] \gg \mathcal{E}[\![F]\!] \\
\mathcal{E}[\![E,F]\!] &= \mathcal{E}[\![F]\!] \circledcirc \mathcal{E}[\![E]\!]
\end{aligned}$$

As an example, the meaning of the grammar $\mu(\lambda S.a"\ ;\ a",S)$ is the relation

$$\mu(\lambda S.(\mathrm{READ}\ a \cup S \circ \mathrm{READ}\ a))$$

and $\mathrm{Br}\ S$ is the list-valued function

$$\mu(\lambda S.(\mathrm{READ}\ a \gg S \circledcirc \mathrm{READ}\ a))$$

The above semantics might be the most intuitive but is not unique. There is different one which is not as obvious.

### 5.3.2 Direct semantics à la Salter and Jones

The semantics of Salter and Jones results from defining elementary relations as the graph of elementary functions. This has the advantage that the peephole laws become applicable, but on the other hand it requires the introduction of a special error input $\Delta$.

$$\begin{aligned}
\mathrm{EMPTY}\ i &= i \\
\mathrm{FAIL}\ i &= \Delta \\
\mathrm{READ}\ C\ (C' \rightarrowtail\!\!+ i) &= (C = C') \rightarrow i \mathbin{[\!]} \Delta \\
\mathrm{EOS}\ i &= (i = [\,]) \rightarrow [\mathit{true}] \mathbin{[\!]} [\,]
\end{aligned}$$

A whole spectrum of possible semantics results when defining some relations as graphs and others as choices. We now continue by transforming the direct semantics of de Vink.

## 5.4 Introducing success continuations

As a first step towards a continuation semantics for $\mathcal{B}$ we introduce *success continuations*. The intuition behind the transformation is to make the implicit evaluation order of $(\sigma \circledcirc f)\ i$ explicit by turning $f \in \mathit{input} \to \mathit{input}*$ into a function $\mathit{Conc}\ f \in \mathit{success} \to \mathit{success}$ which takes $\sigma \in \mathit{success} = \mathit{input} \to \mathbb{B}*$ as an argument that is called after *successful* evaluation of $f$:

$$\mathit{Conc}\ f\ \sigma \quad = \quad \sigma \circledcirc f$$

A left-inverse for $\mathit{Conc}$ is not needed, but note that $\mathit{Conc}$ is strict. The meaning of a complete script is calculated as follows:

$$\begin{array}{cl}
 & \mathcal{M}[\![\textbf{script}\ E]\!] \\
= & \quad \text{unfold} \\
 & \mathrm{E\acute{O}S} \circledcirc \acute{\mathcal{E}}[\![E]\!] \\
= & \quad \text{fold} \\
 & \mathit{Conc}\ \acute{\mathcal{E}}[\![E]\!]\ \mathrm{E\acute{O}S} \\
= & \quad \text{assume fusion } \mathit{Conc}\ \acute{\mathcal{E}}[\![E]\!] = \grave{\mathcal{E}}[\![E]\!] \\
 & \grave{\mathcal{E}}[\![E]\!]\ \mathrm{E\acute{O}S} \\
= & \quad \text{everything needs a continuation} \\
 & \grave{\mathcal{E}}[\![E]\!]\ (\mathrm{E\grave{O}S}\ \bot)
\end{array}$$

where $\mathrm{E\grave{O}S}\ \sigma\ i = (i = [\,]) \to [\mathit{true}]\ [\!]\ [\,]$. For the moment, we do *NOT* assume that the continuation argument is available at compile-time, see §5.6.1.

Specializing the fusion law to verify our wish $\mathit{Conc}\ \acute{\mathcal{E}}[\![E]\!] = \grave{\mathcal{E}}[\![E]\!]$, we get

$$\begin{array}{lcl}
\mathit{Conc}\ \acute{\mathcal{E}}[\![E]\!] = \grave{\mathcal{E}}[\![E]\!] & \Leftarrow & \mathit{Conc}\ \mathrm{FA\acute{I}L} = \mathrm{FA\grave{I}L}\ \wedge \\
 & & \mathit{Conc}\ \mathrm{EM\acute{P}TY} = \mathrm{EM\grave{P}TY}\ \wedge \\
 & & \mathit{Conc}\ (\mathrm{RE\acute{A}D}\ C) = \mathrm{RE\grave{A}D}\ C\ \wedge \\
 & & \mathit{Conc}\ (\mathrm{s\acute{e}q}\ (e, f)) = \mathrm{s\grave{e}q}\ (\mathit{Conc}\ e, \mathit{Conc}\ f)\ \wedge \\
 & & \mathit{Conc}\ (e \acute{\gg} f) = (\mathit{Conc}\ e \grave{\gg} \mathit{Conc}\ f)
\end{array}$$

We start by deriving new versions of the elementary actions. First $\mathrm{FAIL}$ and $\mathrm{EMPTY}$;

$$\begin{array}{cl}
 & \mathit{Conc}\ \mathrm{FA\acute{I}L}\ \sigma\ i \\
= & \quad \text{unfold} \\
 & (\sigma \circledcirc \mathrm{FA\acute{I}L})\ i \\
= & \quad \text{evaluate} \\
 & [\,] \\
= & \quad \text{synthesize} \\
 & \mathrm{FA\grave{I}L}\ \sigma\ i
\end{array}
\qquad\qquad
\begin{array}{cl}
 & \mathit{Conc}\ \mathrm{EM\acute{P}TY}\ \sigma\ i \\
= & \quad \text{unfold} \\
 & (\sigma \circledcirc \mathrm{EM\acute{P}TY})\ i \\
= & \quad \text{evaluate} \\
 & \sigma\ i \\
= & \quad \text{synthesize} \\
 & \mathrm{EM\grave{P}TY}\ \sigma\ i
\end{array}$$

In this way we have found two new instructions $\grave{\mathrm{FAIL}}\ \sigma\ i = [\,]$ and $\grave{\mathrm{EMPTY}}\ \sigma\ i = \sigma\ i$.

For $\mathrm{READ}\ C$ two cases have to be considered.

$$
\begin{aligned}
& \mathit{Conc}\ (\mathrm{RE\acute{A}D}\ C)\ \sigma\ (C \mathrel{\succ\!\!\!+} i) \\
= & \quad \text{unfold} \\
& (\sigma \circledcirc \mathrm{RE\acute{A}D}\ C)\ (C \mathrel{\succ\!\!\!+} i) \\
= & \quad \text{evaluate} \\
& (+\!\!+/ \circ \sigma{*} \circ \mathrm{RE\acute{A}D}\ C)\ (C \mathrel{\succ\!\!\!+} i) \\
= & \quad \text{evaluate} \\
& (+\!\!+/ \circ \sigma{*})\ [i] \\
= & \quad \text{evaluate} \\
& \sigma\ i \\
= & \quad \text{synthesize} \\
& \mathrm{RE\grave{A}D}\ C\ \sigma\ (C \mathrel{\succ\!\!\!+} i)
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Conc}\ (\mathrm{RE\acute{A}D}\ C)\ \sigma\ (C' \mathrel{\succ\!\!\!+} i) \\
= & \quad \text{unfold} \\
& (\sigma \circledcirc \mathrm{RE\acute{A}D}\ C)\ (C \mathrel{\succ\!\!\!+} i) \\
= & \quad \text{evaluate} \\
& (+\!\!+/ \circ \sigma{*} \circ \mathrm{RE\acute{A}D}\ C)\ (C' \mathrel{\succ\!\!\!+} i) \\
= & \quad \text{evaluate} \\
& (+\!\!+\ / \circ \sigma{*})\ [\,] \\
= & \quad \text{evaluate} \\
& [\,] \\
= & \quad \text{synthesize} \\
& \mathrm{RE\grave{A}D}\ C\ \sigma\ (C' \mathrel{\succ\!\!\!+} i)
\end{aligned}
$$

In the above definition of $\mathrm{RE\grave{A}D}\ C$ the purpose of the success continuation $\sigma$ is quite clear; after a symbol has been recognized successfully, evaluation continues with $\sigma$.

For composite actions we have to consider $\mathit{Conc} \circ \circledcirc$ and $\mathit{Conc} \circ \acute{\gg}$.

$$
\begin{aligned}
& \mathit{Conc}\ (e \circledcirc f)\ \sigma \\
= & \quad \text{unfold} \\
& \sigma \circledcirc e \circledcirc f \\
= & \quad \text{fold twice} \\
& \mathit{Conc}\ f\ (\mathit{Conc}\ e\ \sigma) \\
= & \quad \text{synthesize} \\
& (\mathit{Conc}\ f \circ \mathit{Conc}\ e)\ \sigma
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Conc}\ (e \mathrel{\acute{\gg}} f)\ \sigma\ i \\
= & \quad \text{unfold} \\
& (\sigma \circledcirc e \mathrel{\acute{\gg}} f)\ i \\
= & \quad \text{evaluate} \\
& (+\!\!+/ \circ \sigma{*})\ (e\ i +\!\!+ f\ i) \\
= & \quad \text{evaluate} \\
& (\sigma \circledcirc e\ i) +\!\!+ (\sigma \circledcirc f\ i) \\
= & \quad \text{synthesize} \\
& (\mathit{Conc}\ e \mathrel{\grave{\gg}} \mathit{Conc}\ f)\ \sigma\ i
\end{aligned}
$$

Hence the fusion law gives us a new semantics.

$$
\begin{array}{rcl}
\mathcal{M}[\![\_]\!] & \in & \mathit{script} \to (\mathit{input} \to \mathbb{B}{*}) \\
\mathcal{M}[\![\textbf{script}\ E]\!]\ i & = & (\mathrm{EOS} \circledcirc \mathcal{E}[\![E]\!])\ i \\
\mathcal{E}[\![\_]\!] & \in & \mathit{expr} \to (\mathit{success} \to \mathit{success}) \\
\mathcal{E}[\![\textbf{fail}]\!] & = & \grave{\mathrm{FAIL}} \\
\mathcal{E}[\![\textbf{empty}]\!] & = & \grave{\mathrm{EMPTY}} \\
\mathcal{E}[\![C"]\!] & = & \grave{\mathrm{read}}\ C \\
\mathcal{E}[\![E\ ;\ F]\!] & = & \mathcal{E}[\![E]\!] \mathrel{\grave{\gg}} \mathcal{E}[\![F]\!]
\end{array}
$$

$$\mathcal{E}[\![E,F]\!] \quad = \quad \mathcal{E}[\![E]\!] \circ \mathcal{E}[\![F]\!]$$

with semantic actions

$$\begin{aligned}
\mathrm{EOS}\ \sigma\ i &= i = [\,] \to [\mathrm{true}]\ [\!]\ [\,] \\
\mathrm{READ}\ C\ \sigma\ (C' \rightarrowtail\!\!+ i) &= (C = C') \to (\sigma\ i)\ [\!]\ [\,] \\
\mathrm{FAIL}\ \sigma\ i &= [\,] \\
\mathrm{EMPTY}\ \sigma\ i &= \sigma\ i \\
(e \gg f)\ \sigma\ i &= e\ \sigma\ i +\!\!+ f\ \sigma\ i \\
(e \circ f)\ \sigma\ i &= e\ (f\ \sigma)\ i
\end{aligned}$$

## 5.5 Introducing failure continuations

In the previous semantics a choice $e \gg f$ is defined as

$$(e \gg f)\ \sigma\ i \quad = \quad e\ \sigma\ i +\!\!+ f\ \sigma\ i$$

This definition of $\gg$ disguises an implicit evaluation order, first evaluate $e\ \sigma\ i$ then $f\ \sigma\ i$, that needs to be exposed by introducing (failure) continuations, an argument that is called after *unsuccessful* termination of $e\ \sigma\ i$. The function $\mathrm{Conc}$ maps an abstract success continuation $\acute{\sigma}$ into a concrete continuation $\grave{\sigma} = \mathrm{Conc}\ \acute{\sigma}$ that expects a failure continuation.

$$\mathrm{Conc}\ \sigma\ i\ \varphi \quad = \quad \sigma\ i +\!\!+ \varphi$$

The left-inverse $\mathrm{Abs}$ of $\mathrm{Conc}$ takes such a concrete continuation back into an abstract continuation.

$$\begin{aligned}
& \mathrm{Abs}\ \grave{\sigma} \\
= & \quad \grave{\sigma} = \mathrm{Conc}\ \acute{\sigma} \\
& \mathrm{Abs}\ (\mathrm{Conc}\ \acute{\sigma})\ i \\
= & \quad \text{wish} \\
& \acute{\sigma}\ i \\
= & \quad \text{aim at folding } \mathrm{Conc} \\
& \acute{\sigma}\ i +\!\!+ [\,] \\
= & \quad \text{fold } \mathrm{Conc} \\
& \mathrm{Conc}\ \acute{\sigma}\ i\ [\,] \\
= & \quad \mathrm{Conc}\ \acute{\sigma} = \grave{\sigma} \\
& \grave{\sigma}\ i\ [\,]
\end{aligned}$$

From which we conclude that $Abs\ \sigma\ i = \sigma\ i\ [\,]$ is a suitable left-inverse of $Conc$.

The new semantics with failure continuations is derived by exploiting the fact that $Abs \circ Conc = id$.

$$
\begin{array}{ll}
 & \grave{\mathcal{M}}[\![\textbf{script}\ E]\!]\ i \\
= & \quad \text{demand} \\
 & \acute{\mathcal{M}}[\![\textbf{script}\ E]\!]\ i \\
= & \quad \text{unfold} \\
 & \acute{\mathcal{E}}[\![E]\!]\ (E\acute{O}S\ \bot)\ i \\
= & \quad A \circ C = id \\
 & (Abs \circ Conc \circ \acute{\mathcal{E}}[\![E]\!])\ (E\acute{O}S\ \bot)\ i \\
= & \quad \text{unfold} \\
 & (Conc \circ \acute{\mathcal{E}}[\![E]\!])\ (E\acute{O}S\ \bot)\ i\ [\,] \\
= & \quad \text{assume } Conc \circ \acute{\mathcal{E}}[\![E]\!] = \grave{\mathcal{E}}[\![E]\!] \circ Conc \\
 & (\grave{\mathcal{E}}[\![E]\!] \circ Conc)\ (E\acute{O}S\ \bot)\ i\ [\,] \\
= & \quad \text{get rid of } Conc \\
 & \grave{\mathcal{E}}[\![E]\!]\ (E\grave{O}S\ \bot)\ i\ [\,]
\end{array}
$$

where $E\grave{O}S$ is calculated by

$$
\begin{array}{ll}
 & E\grave{O}S\ \sigma\ i\ \varphi \\
= & Conc\ (E\acute{O}S\ \sigma)\ i\ \varphi \\
= & E\acute{O}S\ \sigma\ i +\!\!+\ \varphi \\
= & (i = [\,] \rightarrow [true]\ [\!]\ [\,] +\!\!+\ \varphi)
\end{array}
$$

The necessary conditions to prove the claim $Conc \circ \acute{\mathcal{E}}[\![E]\!] = \grave{\mathcal{E}}[\![E]\!] \circ Conc$ follow from instantiating the induction principle for catamorphisms.

$$
\begin{array}{lll}
Conc \circ \acute{\mathcal{E}}[\![E]\!] = \grave{\mathcal{E}}[\![E]\!] \circ Conc & \Leftarrow & Conc \circ FA\acute{I}L = FA\grave{I}L \circ Conc\ \wedge \\
 & & Conc \circ EM\acute{P}TY = EM\grave{P}TY \circ Conc\ \wedge \\
 & & Conc \circ RE\acute{A}D\ C = RE\grave{A}D\ C \circ Conc\ \wedge \\
 & & (Conc \circ (\acute{e} \gg \acute{f}) = (\grave{e} \gg \grave{f}) \circ Conc \\
 & & \quad \Leftarrow Conc \circ \acute{e} = \grave{e} \circ Conc\ \wedge \\
 & & \quad\quad Conc \circ \acute{f} = \grave{f} \circ Conc)\ \wedge \\
 & & (Conc \circ (\acute{e} \circ \acute{f}) = (\grave{e} \circ \grave{f}) \circ Conc \\
 & & \quad \Leftarrow Conc \circ \acute{e} = \grave{e} \circ Conc\ \wedge \\
 & & \quad\quad Conc \circ \acute{f} = \grave{f} \circ Conc)
\end{array}
$$

The simplest but most satisfactory cases are $EMPTY$ and $FAIL$. Especially the concrete version of $\mathsf{FAIL}$ nicely demonstrates the intuition behind failure continuations being the part of computation that has to be done upon failure.

$$\begin{array}{llll}
 & \mathrm{Conc}\ (\acute{\mathsf{EMPTY}}\ \sigma)\ i\ \varphi & & \mathrm{Conc}\ (\acute{\mathsf{FAIL}}\ \sigma)\ i\ \varphi \\
= & \quad \text{unfold} & = & \quad \text{unfold} \\
 & \acute{\mathsf{EMPTY}}\ \sigma\ i \mathbin{+\!\!+} \varphi & & \acute{\mathsf{FAIL}}\ \sigma\ i \mathbin{+\!\!+} \varphi \\
= & \quad \text{evaluate} & = & \quad \text{evaluate} \\
 & \sigma\ i \mathbin{+\!\!+} \varphi & & [\,] \mathbin{+\!\!+} \varphi \\
= & \quad \text{fold} & = & \quad \text{evaluate} \\
 & \mathrm{Conc}\ \sigma\ i\ \varphi & & \varphi \\
= & \quad \text{synthesize} & = & \quad \text{synthesize} \\
 & \grave{\mathsf{EMPTY}}\ (\mathrm{Conc}\ \sigma)\ i\ \varphi & & \grave{\mathsf{FAIL}}\ (\mathrm{Conc}\ \sigma)\ i\ \varphi
\end{array}$$

Hence we have determined $\grave{\mathsf{EMPTY}}\ \sigma\ i\ \varphi = \sigma\ i\ \varphi$ and $\grave{\mathsf{FAIL}}\ \sigma\ i\ \varphi = \varphi$.

Just like for $\mathcal{W}$, the naive choice of only requiring that $\mathrm{Conc}\ \acute{\mathcal{E}}[\![E]\!] = \grave{\mathcal{E}}[\![E]\!]$ would give an unsatisfactory semantics. For $\mathsf{EMPTY}$ this amounts to showing $\mathrm{Conc}\ (\acute{\mathsf{EMPTY}}\ \sigma) = \grave{\mathsf{EMPTY}}\ \sigma$

$$\begin{array}{ll}
 & \mathrm{Conc}\ (\acute{\mathsf{EMPTY}}\ \sigma)\ i\ \varphi \\
= & \quad \text{unfold} \\
 & \acute{\mathsf{EMPTY}}\ \sigma\ i \mathbin{+\!\!+} \varphi \\
= & \quad \text{evaluate} \\
 & \sigma\ i \mathbin{+\!\!+} \varphi \\
= & \quad \text{synthesize} \\
 & \grave{\mathsf{EMPTY}}\ \sigma\ i\ \varphi
\end{array}$$

Taking $\grave{\mathsf{EMPTY}}\ \sigma\ i\ \varphi = \sigma\ i \mathbin{+\!\!+} \varphi$ is not a sensible realization of an instruction that ‘does nothing’.

For $\mathsf{READ}\ C$ we have to check two cases, one boils down to $\mathsf{EMPTY}$, the other to $\mathsf{FAIL}$.

$$
\begin{array}{llcll}
 & \mathrm{Conc}\ (\mathrm{RE\acute{A}D}\ C\ \sigma)\ (C \mathrel{\succ\!\!+} i) & \qquad & & \mathrm{Conc}\ (\mathrm{RE\acute{A}D}\ C\ \sigma)\ (C' \mathrel{\succ\!\!+} i) \\
= & \quad \textsf{unfold} & & = & \quad \textsf{unfold} \\
 & \mathrm{RE\acute{A}D}\ C\ \sigma\ (C \mathrel{\succ\!\!+} i) \mathbin{+\!\!+} \varphi & & & \mathrm{RE\acute{A}D}\ C\ \sigma\ (C' \mathrel{\succ\!\!+} i) \mathbin{+\!\!+} \varphi \\
= & \quad \textsf{evaluate} & & = & \quad \textsf{evaluate} \\
 & \sigma\ i \mathbin{+\!\!+} \varphi & & & [\,] \mathbin{+\!\!+} \varphi \\
= & \quad \textsf{fold} & & = & \quad \textsf{evaluate} \\
 & \mathrm{Conc}\ \sigma\ i\ \varphi & & & \varphi \\
= & \quad \textsf{synthesize} & & = & \quad \textsf{synthesize} \\
 & \mathrm{RE\grave{A}D}\ C\ (\mathrm{Conc}\ \sigma)\ (C \mathrel{\succ\!\!+} i)\ \varphi & & & \mathrm{RE\grave{A}D}\ C\ (\mathrm{Conc}\ \sigma)\ (C' \mathrel{\succ\!\!+} i)\ \varphi
\end{array}
$$

For finding the concrete version of alternation and sequencing the induction hypothesis must be used.

$$
\begin{array}{llcll}
 & \mathrm{Conc}\ ((\acute{e} \mathbin{\acute{\gg}} \acute{f})\ \sigma)\ i\ \varphi & \qquad & & \\
= & \quad \textsf{unfold} & & & \\
 & \acute{e}\ \sigma\ i \mathbin{+\!\!+} \acute{f}\ \sigma\ i \mathbin{+\!\!+} \varphi & & & \mathrm{Conc}\ (\acute{e}\ (\acute{f}\ \sigma)) \\
= & \quad \textsf{fold} & & = & \quad \textsf{IH} \\
 & \mathrm{Conc}\ (\acute{e}\ \sigma)\ i\ (\mathrm{Conc}\ (\acute{f}\ \sigma)\ i\ \varphi) & & & \grave{e}\ (\mathrm{Conc}\ (\acute{f}\ \sigma)) \\
= & \quad \textsf{IH twice} & & = & \quad \textsf{IH} \\
 & \grave{e}\ (\mathrm{Conc}\ \sigma)\ i\ (\grave{f}\ (\mathrm{Conc}\ \sigma)\ i\ \varphi) & & & \grave{e}\ (\grave{f}\ (\mathrm{Conc}\ \sigma)) \\
= & \quad \textsf{synthesize} & & & \\
 & (\grave{e} \mathbin{\grave{\gg}} \grave{f})\ (\mathrm{Conc}\ \sigma)\ i\ \varphi & & &
\end{array}
$$

### 5.5.1 Recapitulation: Continuation style semantics

At this moment the semantic domains are defined as (when transforming the semantics further the types $success$ and $failure$ will change accordingly):

$$
\begin{array}{rcl}
e, f, g \in action & = & success \to success \\
\sigma \in success & = & input \to failure \to \mathbb{B}* \\
\varphi \in failure & = & \mathbb{B}* \\
recognizer & = & input \to \mathbb{B}*
\end{array}
$$

Our semantics employs two continuations, $\sigma \in success$ and $\varphi \in failure$. Computation of $f\ \sigma\ i\ \varphi$ proceeds according to $\sigma$ if $f \in action$ terminates successfully, and according to $\varphi$ if $f$ fails. Thinking in terms of extending our backtrack language to full Prolog, the input $i$ can be thought of holding the values or bindings of variables.

$$
\begin{array}{rcl}
\mathrm{EMPTY}\ \sigma\ i\ \varphi & = & \sigma\ i\ \varphi \\
\mathrm{FAIL}\ \sigma\ i\ \varphi & = & \varphi
\end{array}
$$

$$\begin{aligned} \mathrm{READ}\ C\ \sigma\ (C' \rightarrowtail i)\ \varphi &= (C = C') \to \sigma\ i\ \varphi \ [\!]\ \varphi && (5.1)\\ (e \gg f)\ \sigma\ i\ \varphi &= e\ \sigma\ i\ (f\ \sigma\ i\ \varphi) && (5.2)\\ \mathrm{EOS}\ \sigma\ i\ \varphi &= (i = [\,] \to [\mathrm{true}]\ [\!]\ [\,]) +\!\!+\ \varphi \end{aligned}$$

Evaluating a choice $(e \gg f)\ \sigma\ i\ \varphi$ effectively 'pushes' a new alternative on the stack $\varphi$ of open alternatives; $\varphi' = f\ \sigma\ i\ \varphi$. An failing READ C or an explicit FAIL amounts to executing a jump to the last open alternative on top of the stack. If no such alternative exists then the whole computation ends in failure. The action EOS checks whether the complete input has been recognized and tries the next open alternative after signalling success.

The translation scheme is given by $\mathcal{E}[\![\_]\!] = (\!|\mathrm{EMPTY}, \mathrm{FAIL}, \mathrm{READ}, \circ, \gg|\!)$, with $\mathcal{M}[\![\_]\!] = (\!|parse|\!)$ where $parse\ e\ i = e\ (\mathrm{EOS}\ \bot)\ i\ [\,]$. An alternative definition we will use is $parse\ e\ i = (e \gg \mathrm{DONE})\ (\mathrm{EOS}\ \bot)\ i\ \bot$ where $\mathrm{DONE}\ \sigma\ i\ \varphi = [\,]$. This latter definition has the advantage that it is defined entirely in terms of functions, no lists appear in this definition of $parse$.

### 5.5.2 Algebraic properties

From either of the above semantics we may derive some (rather obvious) algebraic properties for $\mathcal{B}$. The neutral element of sequencing is EMPTY, while the neutral element of choice is FAIL.

$$\begin{aligned} \mathrm{EMPTY} \circ e &= e & \qquad e \gg \mathrm{FAIL} &= e\\ e \circ \mathrm{EMPTY} &= e & \qquad \mathrm{FAIL} \gg e &= e \end{aligned}$$

Finally, FAIL is the left absorbing element of sequencing.

$$\mathrm{FAIL} \circ e = \mathrm{FAIL}$$

The fact that sequencing distributes from the right over choice is easily proved.

$$(e \gg f) \circ g = (e \circ g) \gg (f \circ g)$$

Surprisingly (?) distribution from the left, $e \circ (f \gg g) = (e \circ f) \gg (e \circ g)$ does not hold. A counter example is obtained by taking $g = \bot$; left factorization is not a safe grammatical transformation in the context of backtrack parsers.

## 5.6 Making recursion explicit

In order to transform our semantics even further recursion in the source language explicit must be made explicit. This is done by introducing *nonterminal* expressions **call** $expr$.

$$\begin{aligned} script &::= expr\\ expr &::= \textbf{empty} \mid \textbf{fail} \mid symbol" \mid \textbf{call}\ expr \mid expr\ \textbf{;}\ expr \mid expr\textbf{,}expr \end{aligned}$$

Recursive expressions are still represented by cyclic programs, but with the constraint that each cycle is broken by a **call**. There is no way to warrant this by refining the syntax. Instead of writing $\mu(\lambda S.a"\,;\,a",S)$ we should use $\mu(\lambda S.a"\,;\,a",\textbf{call}\ S)$

### 5.6.1 Goal Stacking semantics

The translation of expressions is extended to deal with calls.

$$\mathcal{E}[\![\textbf{call}\ E]\!] \quad = \quad \mathrm{CALL}\ \mathcal{E}[\![E]\!]$$

Informally calling an nonterminal **call** $E$ means *pushing* the code for its body $\mathcal{E}[\![E]\!]$ onto the success continuation $\sigma$. Therefore we define the associated action by

$$\mathrm{CALL}\ e\ \sigma\ i\ \varphi \quad = \quad (e\ \sigma)\ i\ \varphi$$

In the theorem proving literature this is known as *goal stacking*. The definition of CALL makes clear why we cannot assume the success continuation to be available at compile-time. That would mean that there is no operational difference with the previous semantics without CALL. A concrete implementation based on goal stacking must use *structure copying* for operator bodies. This approach will be pursued in section 5.10.1, leading to the *Recursive Backup Machine*.

### 5.6.2 Goal Jumping

Now consider that before evaluating an operator **call** $E$, the current success continuation is saved for later resumption on a *dump* continuation $\delta$. Then the body $E$ is evaluated (hence it must be of type $success$). When this evaluation is finished, control returns to the previous continuation restored from the dump. Using this scheme, the success continuation becomes available at compile-time. Under this scenario the success continuation is only traversed but never modified so that *structure sharing* can be used. This development will lead to the *Warren Abstract Machine*. We won't introduce the dump continuation formally, but it would cause no real trouble if we would wish to do so.

The semantic domains for the goal jumping semantics are:

$$\begin{aligned}
e, f, g \in action &= success \to success \\
\sigma \in success &= dump \to dump \\
\delta \in dump &= input \to failure \to \mathbb{B}* \\
\varphi \in failure &= \mathbb{B}*
\end{aligned}$$

According to the informal explanation we redefine the run-time operator CALL and define RETURN as:

$$\begin{aligned}
\mathrm{CALL}\ \sigma'\ \sigma\ \delta\ i\ \varphi &= \sigma'\ (\sigma\ \delta)\ i\ \varphi \\
\mathrm{RETURN}\ \sigma\ \delta\ i\ \varphi &= \delta\ i\ \varphi
\end{aligned}$$

Apart from taking an extra argument, the remaining operators remain unchanged:

$$\begin{array}{rcl}
\mathrm{EMPTY}\ \sigma\ \delta\ i\ \varphi & = & \sigma\ \delta\ i\ \varphi \\
\mathrm{FAIL}\ \sigma\ \delta\ i\ \varphi & = & \varphi \\
\mathrm{READ}\ C\ \sigma\ \delta\ (C' \succ\!\!+ i)\ \varphi & = & (C = C') \rightarrow (\sigma\ \delta\ i\ \varphi)\ [\!]\ \varphi \\
(e \gg f)\ \sigma\ \delta\ i\ \varphi & = & e\ \sigma\ \delta\ i\ (f\ \sigma\ \delta\ i\ \varphi) \\
\mathrm{EOS}\ \sigma\ \delta\ i\ \varphi & = & (i = [\,] \rightarrow [true]\ [\!]\ [\,]) +\!\!+\ \varphi
\end{array}$$

The meaning of a script is now given by:

$$\begin{array}{rcl}
\mathcal{M}[\![\textbf{script}\ E]\!]\ i & = & \mathcal{E}[\![E]\!]\ (\mathrm{EOS}\ \bot)\ \bot\ i\ [\,] \\
\mathcal{E}[\![\textbf{fail}]\!] & = & \mathrm{FAIL} \\
\mathcal{E}[\![\textbf{empty}]\!] & = & \mathrm{EMPTY} \\
\mathcal{E}[\![C"]\!] & = & \mathrm{READ}\ C \\
\mathcal{E}[\![\textbf{call}\ E]\!] & = & \mathrm{CALL}\ (\mathcal{E}[\![E]\!]\ (\mathrm{RETURN}\ \bot)) \\
\mathcal{E}[\![E\ ;\ F]\!] & = & \mathcal{E}[\![E]\!] \gg \mathcal{E}[\![F]\!] \\
\mathcal{E}[\![E,F]\!] & = & \mathcal{E}[\![E]\!] \circ \mathcal{E}[\![F]\!]
\end{array}$$

### 5.6.3 Refining alternation

A nice property of the goal jumping semantics is that for each choice $(e \gg f)\ \sigma$, the success continuation $\sigma$ is known at compile-time. This allows the following optimization:

$$\begin{array}{ll}
 & (e \gg f)\ \sigma\ \delta\ i\ \varphi \\
= & e\ \sigma\ \delta\ i\ (f\ \sigma\ \delta\ i\ \varphi) \\
= & \mathrm{TRY\_ME\_ELSE}\ (f\ \sigma)\ (e\ \sigma)\ \delta\ i\ \varphi
\end{array}$$

This can be realized by changing the compilation functions into

$$\begin{array}{rcl}
\mathcal{M}[\![\textbf{script}\ E]\!]\ i & = & \mathcal{E}[\![E]\!]\ (\mathrm{EOS}\ \bot)\ \bot\ i\ [\,] \\
\mathcal{E}[\![\textbf{empty}]\!]\ \sigma & = & \sigma \\
\mathcal{E}[\![\textbf{fail}]\!]\ \sigma & = & \mathrm{FAIL}\ \sigma \\
\mathcal{E}[\![C"]\!]\ \sigma & = & \mathrm{READ}\ C\ \sigma \\
\mathcal{E}[\![\textbf{call}\ E]\!]\ \sigma & = & \mathrm{CALL}\ (\mathcal{E}[\![E]\!]\ (\mathrm{RETURN}\ \bot))\ \sigma \\
\mathcal{E}[\![E\ ;\ F]\!]\ \sigma & = & \mathrm{TRY\_ME\_ELSE}\ (\mathcal{E}[\![F]\!]\ \sigma)\ (\mathcal{E}[\![E]\!]\ \sigma) \\
\mathcal{E}[\![E,F]\!]\ \sigma & = & \mathcal{E}[\![E]\!]\ (\mathcal{E}[\![F]\!]\ \sigma)
\end{array}$$

As the more symmetric $(e \gg f)$ is easier in proofs, we will not use this refined choice semantics until we start developing the WAM.

### 5.6.4 Tail call elimination

*Tail call elimination* is an extremely important optimization, without this optimization the time complexity for the definition $\mu(\lambda S.a$",**call** $S$ **; empty**), would become quadratic instead of linear in the length of the input [62]. Tail-call elimination simply follows from the calculation

$$\begin{array}{ll} & \mathrm{CALL}\ \sigma'\ (\mathrm{RETURN}\ \sigma)\ \delta\ i\ \varphi \\ = & \sigma'\ ((\mathrm{RETURN}\ \sigma)\ \delta)\ i\ \varphi \\ = & \sigma'\ \delta\ i\ \varphi \\ = & \mathrm{JUMP}\ \sigma'\ \sigma\ \delta\ i\ \varphi \end{array}$$

which proves that we may replace CALL $\sigma \circ$ RETURN by JUMP $\sigma$. Tail call optimization is a nice example of two functions that are declaratively equal but operationally drastically different. Under a more operational interpretation, we may read tail call elimination as "a subroutine call immediately followed by a return may be be replaced by an ordinary jump". One of the strong points of our approach is that important optimizations often follow by expanding a few definitions and then just calculating. As we have the success continuation available at compile time, we could modify $\mathcal{E}[\![\_]\!]$ such that tail call elimination is done directly.

#### An implementation pitfall

An obvious implementation for the current semantics would be to define each instruction as a C [53] procedure:

```
typedef void (*cont) (void);
void JUMP (cont sigma){
   (*sigma)();
   };
```

However since most C-compilers do not do tail-call optimization for the call `(*sigma)();` the resulting implementation will still manifest a quadratic behavior ! The *m88k* code generated by GCC on the *DG Aviion* for `JUMP` for example is

```
_JUMP:
    subu     r31,r31,0x0028        ; 40
    st       r1,r31,0x0024         ; 36
@Locs.a0:
    jsr      r2
@Locs.b0:
    ld       r1,r31,0x0024         ; 36
    jmp.n    r1
    addu     r31,r31,0x0028        ; 40
```

whereas the required code is simply `_JUMP: jmp r2` .

**Example**

Using both tail-call elimination and refined choice, the grammar $\mu(\lambda S.a" \mathbin{;} a",a" \mathbin{;} a",\textbf{call}\ S \mathbin{;} a",a",\textbf{call}\ S)$ compiles into:

$$\begin{aligned}
S &= (\mathrm{TRY\_ME\_ELSE}\ S' \circ \mathrm{READ}\ a \circ \mathrm{RETURN})\ \bot \\
S' &= (\mathrm{TRY\_ME\_ELSE}\ S'' \circ \mathrm{READ}\ a \circ \mathrm{READ}\ a \circ \mathrm{RETURN})\ \bot \\
S'' &= (\mathrm{TRY\_ME\_ELSE}\ S''' \circ \mathrm{READ}\ a \circ \mathrm{JUMP}\ S)\ \bot \\
S''' &= (\mathrm{READ}\ a \circ \mathrm{READ}\ a \circ \mathrm{JUMP}\ S)\ \bot
\end{aligned}$$

In WAM parlance, RETURN means $\mathit{proceed}$, JUMP means $\mathit{execute}$ and CALL means $\mathit{call}$. The *local stack* of the WAM is represented by the dump and the failure continuation. The input abstracts away from the *global stack* and the *trail*. The READ instruction represents the *get*-, *put*-, and *indexing* instruction classes. After introducing the cut, we will show how the dump and failure continuations can be defunctionalized into linked lists which then can be merged into a single stack. We then also show how TRY_ME_ELSE can be refined into the triple TRY_ME_ELSE,RETRY_ME_ELSE and TRUST_ME_ELSE_FAIL.

## 5.7 Introducing the ‘cut’

In this section we will extend $\mathcal{B}$ with the notorious *cut* operator, the semantics of which are most easily described using a dump since it provides us with entrance and exit points in the evaluation of operator bodies.

The intended meaning of the **!** operator is to restore the failure continuation that was active at the time the operator was called in whose body the cut appears. Hence, evaluation of a **!** discards the alternatives that have been generated since the body in which the cut appears has been entered. An obvious way to obtain this behavior is by adding an extra argument, the cut continuation $\phi$ which is that old failure continuation. So now we have as semantic domains:

$$\begin{aligned}
\sigma', \sigma \in \mathit{success} &= \mathit{cut} \to \mathit{dump} \to \mathit{dump} \\
\delta \in \mathit{dump} &= \mathit{input} \to \mathit{failure} \to \mathbb{B}* \\
\varphi \in \mathit{failure} &= \mathbb{B}* \\
\phi \in \mathit{cut} &= \mathit{failure}
\end{aligned}$$

Executing a CUT amounts to restoring the failure continuation $\phi$ as it was at the moment of entering the current operator, thereby ‘cutting’ all choices $\varphi$ that were made since.

$$\begin{aligned}
\mathcal{E}[\![\mathbf{!}]\!] &= \mathrm{CUT} \\
\mathrm{CUT}\ \sigma\ \phi\ \delta\ i\ \varphi &= \sigma\ \phi\ \delta\ i\ \phi
\end{aligned}$$

When an operator is called we save the success continuation as well as the current cut continuation on the dump, and set the current cut continuation to the failure continuation. Upon exit the success and cut continuation are restored as they existed just before the call.

$$\begin{aligned} \mathrm{CALL}\ \sigma'\ \sigma\ \phi\ \delta\ i\ \varphi &= \sigma'\ \varphi\ (\sigma\ \phi\ \delta)\ i\ \varphi \\ \mathrm{RETURN}\ \sigma\ \phi\ \delta\ i\ \varphi &= \delta\ i\ \varphi \end{aligned}$$

For operators whose bodies do not contain **!**'s, saving and restoring the cut continuation is not needed. The dump can be interpreted as a list of pairs $(success, failure)$ or equivalently as a pair of lists of respective types. We will use this intuition in the next section when presenting a more efficient implementation for CUT.

The remaining equations essentially remain the same, taking care of the additional cut continuation.

$$\begin{aligned} \mathrm{EMPTY}\ \sigma\ \phi\ \delta\ i\ \varphi &= \sigma\ \phi\ \delta\ i\ \varphi \\ \mathrm{FAIL}\ \sigma\ \phi\ \delta\ i\ \varphi &= \varphi \\ \mathrm{READ}\ C\ \sigma\ \phi\ \delta\ (C' \succ\!\!\!+ i)\ \varphi &= (C = C') \rightarrow (\sigma\ \phi\ \delta\ i\ \varphi) \,[\!]\, \varphi \\ (e \gg f)\ \sigma\ \phi\ \delta\ i\ \varphi &= e\ \sigma\ \phi\ \delta\ i\ (f\ \sigma\ \phi\ \delta\ i\ \varphi) \\ \mathrm{EOS}\ \sigma\ \phi\ \delta\ i\ \varphi &= (i = [\,] \rightarrow [true] \,[\!]\, [\,]) +\!\!+\ \varphi \end{aligned}$$

The meaning of a script is now defined as:

$$\mathcal{M}[\![E]\!]\ i \;=\; \mathcal{E}[\![E]\!]\ (\mathrm{EOS}\ \bot)\ \bot\ \bot\ i\ [\,]$$

We still have that CALL $\sigma' \circ \mathrm{RETURN} = \mathrm{JUMP}\ \sigma'$ holds, but now

$$\mathrm{JUMP}\ \sigma'\ \sigma\ \phi\ \delta\ i\ \varphi \;=\; \sigma'\ \varphi\ \delta\ i\ \varphi$$

### 5.7.1 Alternative definition for !

In [32] an alternative implementation for **!** is given. Instead of saving/restoring the failure continuation on function entry/exit, any expression whose choice-points are to be cut can be bracketed. This allows one to use cuts without the overhead of the associated function call. In order to model this we add a new production to our grammar:

$$\begin{aligned} expr\ &::= \ \ldots \\ &\ \ \mid\ \ \{expr\} \\ &\ \ \mid\ \ \ldots \end{aligned}$$

Informally the meaning of the expression E,{F},G is the same as E,**call** F,G. but without a runtime function call. Hence CALL expressions need only be used for recursive rules. Formally the meaning of this new construct is given by:

$$\mathcal{E}[\![\{E\}]\!] \;=\; \mathrm{SAVECP} \circ \mathcal{E}[\![E]\!] \circ \mathrm{CUTTO}$$

The action $\mathit{SAVECP}$ saves the current failure continuation on top of the cut *stack*, as before $\mathit{CUT}$ replaces the current failure continuation by the failure continuation on top of the cut stack, while the action $\mathit{CUTTO}$ pops the topmost element from the cut stack. Since we partitioned the dump into two stacks this requires that $\phi \in cut = failure*$.

$$\begin{aligned}
\mathit{SAVECP}\ \sigma\ \phi\ \delta\ i\ \varphi &= \sigma\ (\varphi \rightarrowtail \phi)\ \delta\ i\ \varphi \\
\mathit{CUT}\ \sigma\ (\varphi \rightarrowtail \phi)\ \delta\ i\ \varphi' &= \sigma\ (\varphi \rightarrowtail \phi)\ \delta\ i\ \varphi \\
\mathit{CUTTO}\ \sigma\ (\varphi \rightarrowtail \phi)\ \delta\ i\ \varphi &= \sigma\ \phi\ \delta\ i\ \varphi
\end{aligned}$$

### 5.7.2 Some folk theorems

To illustrate our semantics for **!**, we prove some folk theorems [31, 9] from the Prolog community. The first two follow directly from the fact that $\mathit{CUT}\ \sigma\ \varphi\ \delta\ i\ \varphi = \sigma\ \varphi\ \delta\ i\ \varphi$.

$$\mathcal{E}[\![\mathbf{!}\ ,\ \mathbf{!}]\!] = \mathcal{E}[\![\mathbf{!}]\!] \qquad \text{(Folk1)}$$

$$\mathcal{E}[\![\mathbf{call}\ (E\ ;\ (\mathbf{!}\ ,\ F))]\!] = \mathcal{E}[\![\mathbf{call}\ (E\ ;\ F)]\!] \qquad \text{(Folk2)}$$

The following more interesting ones allow systematical introduction and removal of cuts.

An action $e$ is called *determinate* if it can succeed in at most one way, more formally:

- $e$ does not terminate, $e\ \sigma\ \varphi\ \delta\ i\ \varphi = \bot$, or
- $e$ fails, $e\ \sigma\ \varphi\ \delta\ i\ \varphi = \mathit{FAIL}$, or
- $e$ succeeds but creates no choices, $e\ \sigma\ \phi\ \delta\ i\ \varphi = \sigma\ \phi\ \delta'\ i'\ \varphi$

If action $e$ is determinate it is not difficult to show (for $e\ \sigma\ \varphi\ \delta\ i\ \varphi = \bot$ and $e\ \sigma\ \varphi\ \delta\ i\ \varphi = \mathit{FAIL}$ the proof is trivial) that:

$$\begin{aligned}
& (e \circ \mathit{CUT})\ \sigma\ \varphi\ \delta\ i\ \varphi \\
= \ & \mathit{CUT}\ \sigma\ \varphi\ \delta'\ i'\ \varphi \\
= \ & \sigma\ \varphi\ \delta'\ i'\ \varphi \\
= \ & e\ \sigma\ \varphi\ \delta\ i\ \varphi
\end{aligned}$$

The third folk theorem is direct from this property.

$$\mathcal{E}[\![\mathbf{call}\ (E\ ;\ F,\ \mathbf{!}\ ,G)]\!] = \mathcal{E}[\![\mathbf{call}\ (E\ ;\ F,G)]\!] \Leftarrow \mathcal{E}[\![F]\!]\ \text{determinate} \qquad \text{(Folk3)}$$

This theorem is useful to make tail-call elimination applicable.

A choice $e \gg f$ is called *deterministic* if

- $e = \bot$, or

- $e = \mathrm{FAIL}$, or
- $e$ is determinate and $f = \mathrm{FAIL}$.

For deterministic choice we have the following theorem

$$\mathcal{E}[\![\textbf{call}\ (\mathrm{E}\ \textbf{;}\ \mathrm{F})]\!] = \mathcal{E}[\![\textbf{call}\ (\mathrm{E}\textbf{,!}\ \textbf{;}\ \mathrm{F})]\!] \quad \Leftarrow \quad \mathcal{E}[\![\mathrm{E}\ \textbf{;}\ \mathrm{F}]\!]\ \text{deterministic} \qquad \text{(Folk4)}$$

This theorem is particularly useful in cases like **call** (0" **;** . . . **;** 9") which according to (Folk4) can be replaced by the much more efficient **call** (0"**,!** **;** . . . **;** 8"**,!** **;** 9") The most intersting case in the proof is when $f = \mathcal{E}[\![\mathrm{F}]\!] = \mathrm{FAIL}$ and $e = \mathcal{E}[\![\mathrm{E}]\!]$ succeeds determinately.

$$\begin{array}{l} (e \gg f)\ \sigma\ \varphi\ \delta\ i\ \varphi \\ = \\ e\ \sigma\ \varphi\ \delta\ i\ \varphi \\ = \\ (e \circ \mathrm{CUT})\ \sigma\ \varphi\ \delta\ i\ \varphi \\ = \\ ((e \circ \mathrm{CUT}) \gg f)\ \sigma\ \varphi\ \delta\ i\ \varphi \end{array}$$

Note that if we would also allow the transformation in case $f = \perp$ then evaluation of $(e \circ \mathrm{CUT}) \gg f$ might terminate where $e \gg f$ does not.

Since the introduction of the cut has no fundamental influence on the derivations that follow, we shall not take it into account in the remainder of this chapter.

## 5.8 Comparison with other continuation semantics

Two other denotational semantics for $\mathcal{B}$ containing cut based on continuation semantics are those by de Bruin and de Vink [26] and by de Bakker [24]. The relevant part of the former one can be stated in our notation as follows:

$$\begin{array}{rcl} \mathrm{CUT}\ \sigma\ \phi\ i\ \varphi & = & \sigma\ \phi\ i\ \phi \\ \mathrm{CALL}\ e\ \sigma\ \phi\ i\ \varphi & = & e\ (\lambda\phi'.\sigma\ \phi)\ \varphi\ i\ \varphi \\ \mathrm{FAIL}\ \sigma\ \phi\ i\ \varphi & = & \varphi \\ (e \gg f)\ \sigma\ \phi\ i\ \varphi & = & e\ \sigma\ \phi\ i\ (f\ \sigma\ \phi\ i\ \varphi) \end{array}$$

This semantics makes no use of a dump continuation, but encodes successful termination of CALL $e$ by passing $\lambda\phi'.\sigma\ \phi$ as success continuation instead of just $\sigma$, and therefore is well suited for transformations towards a recomputing implementation as will be given in §5.10. We should not forget to mention that this semantics was the source of inspiration of our semantics.

The denotational semantics of de Bakker [24] can can be considered as an intermediate between ours and that of de Bruin and de Vink [26]. Though it also employs a dump, successful termination of CALL $e$ is encoded directly in the success continuation instead of using RETURN. Denotationally this makes no difference as RETURN $\sigma = \mathit{id}$.

$$\begin{aligned}
\mathrm{CUT}\ \sigma\ \phi\ \delta\ i\ \varphi &= \sigma\ \phi\ \delta\ i\ \phi \\
\mathrm{CALL}\ e\ \sigma\ \phi\ \delta\ i\ \varphi &= e\ (\lambda\phi.\lambda\delta.\delta)\ \varphi\ (\sigma\ \phi\ \delta)\ i\ \varphi \\
\mathrm{FAIL}\ \sigma\ \phi\ \delta\ i\ \varphi &= \varphi \\
(e \gg f)\ \sigma\ \phi\ \delta\ i\ \varphi &= e\ \sigma\ \phi\ \delta\ i\ (f\ \sigma\ \phi\ \delta\ i\ \varphi)
\end{aligned}$$

We believe that because our semantics is more structured than the above, it is both more readable as well as more amenable to further transformations.

## 5.9 Concrete Semantics

In order to explain the Warren Abstract Machine [110], we will work with the concrete semantics of expressions. As a first step, all pointers implicit in this representation will be made explicit by specifying the abstract machine instructions using *update schemes*. Next we show how the dump and the failure continuation can be merged into a single stack, called the *local stack* by Warren. Measurements however indicate that a split stack architecture seems to be superior to the single stack model (reduced locality, increased complexity). Moreover we think that Appel's principle [3] holds here as well; the dump and failure continuations should be heap- instead of stack allocated.

The defunctionalized continuations are described by the following algebraic data types.

$$\begin{aligned}
\sigma \in \mathit{Success} &::= \mathrm{EMPTY}\ \mathit{Success} \mid \mathrm{FAIL}\ \mathit{Success} \\
&\mid\ \ \mathrm{READ}\ \mathit{symbol}\ \mathit{Success} \\
&\mid\ \ \mathrm{CALL}\ \mathit{Success} \mid \mathrm{RETURN}\ \mathit{Success} \\
&\mid\ \ \mathrm{TRY\_ME\_ELSE}\ \mathit{Success} \\
&\mid\ \ \mathrm{EOS}\ \mathit{Success} \mid \mathrm{DONE}\ \mathit{Success} \\
\varphi \in \mathit{Failure} &::= \mathrm{CP}\ \mathit{Success}\ \mathit{Dump}\ \mathit{input}\ \mathit{Failure} \\
\delta \in \mathit{Dump} &::= \mathrm{DF}\ \mathit{Success}\ \mathit{Dump}
\end{aligned}$$

As explained in Chapter 1, compiling a script is defined as

$$\mathit{compile}\ (\textbf{script}\ E) \quad = \quad \mathrm{TRY\_ME\_ELSE}\ (\mathrm{DONE}\ \bot)\ (\mathcal{E}[\![E]\!]\ (\mathrm{EOS}\ \bot))$$

where $\mathcal{E}[\![\_]\!] \in \mathit{expr} \to \mathit{Success} \to \mathit{Success}$ is similar to the semantics given in §5.6.3 except that instead of using actions it uses the above defined constructors. When confusion can arise whether symbols should be read as constructors or actions, we use grave accents for constructors and acute accents for actions. Completing the compiler correctness diagram

$$\begin{tikzcd}
\mathit{script} \arrow[rr, "c"] \arrow[dr, "\mathcal{M}[\![\ ]\!]"'] & & \mathit{Success} \arrow[dl, "\lambda\sigma.\lambda i.\mathrm{PC}\ \sigma\ \bot\ i\ \bot"] \\
& \mathit{recognizer} &
\end{tikzcd}$$

automatically gives an *interpreter* for $\mathit{Success}$ programs. Given

$$\mathrm{PC}\ \sigma\ \delta\ i\ \varphi \quad = \quad \mathcal{S}[\![\sigma]\!]\ \mathcal{D}[\![\delta]\!]\ i\ \mathcal{F}[\![\varphi]\!]$$

where $\mathcal{S}[\![\_]\!], \mathcal{D}[\![\_]\!]$ and $\mathcal{F}[\![\_]\!]$ map $\mathit{Success}$ to $\mathit{success}$, $\mathit{Dump}$ to $\mathit{dump}$ and $\mathit{Fail}$ to $\mathit{fail}$ respectively, we can derive by simple 'unfold-simplify-fold' a version of PC that does not use $\mathcal{S}[\![\_]\!], \mathcal{D}[\![\_]\!]$ or $\mathcal{F}[\![\_]\!]$. We only show one step of this derivation.

$$\begin{aligned}
& \mathrm{PC}\ (\grave{\mathrm{EMPTY}}\ \sigma)\ \delta\ i\ \varphi \\
= \quad & \quad \text{unfold} \\
& \mathcal{S}[\![\grave{\mathrm{EMPTY}}\ \sigma]\!]\ \mathcal{D}[\![\delta]\!]\ i\ \mathcal{F}[\![\varphi]\!] \\
= \quad & \quad \text{unfold} \\
& \acute{\mathrm{EMPTY}}\ \mathcal{S}[\![\sigma]\!]\ \mathcal{D}[\![\delta]\!]\ i\ \mathcal{F}[\![\varphi]\!] \\
= \quad & \quad \text{evaluate} \\
& \mathcal{S}[\![\sigma]\!]\ \mathcal{D}[\![\delta]\!]\ i\ \mathcal{F}[\![\varphi]\!] \\
= \quad & \quad \text{fold} \\
& \mathrm{PC}\ \sigma\ \delta\ i\ \varphi
\end{aligned}$$

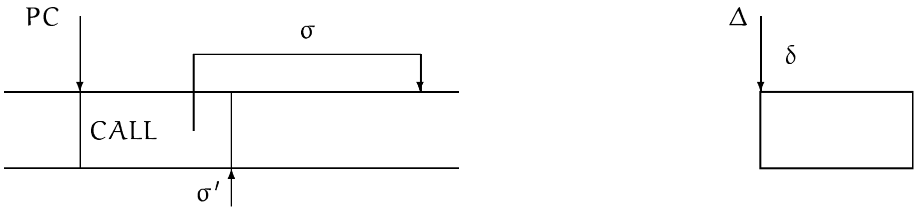
The complete case distinction made by PC is given below.

$$\begin{aligned}
\mathrm{PC}\ (\mathrm{EMPTY}\ \sigma)\ \delta\ i\ \varphi &= \mathrm{PC}\ \sigma\ \delta\ i\ \varphi \\
\mathrm{PC}\ (\mathrm{FAIL}\ \sigma)\ \delta\ i\ (\mathrm{CP}\ \sigma'\ \delta'\ i'\ \varphi) &= \mathrm{PC}\ \sigma'\ \delta'\ i'\ \varphi \\
\mathrm{PC}\ (\mathrm{READ}\ C)\ \sigma\ \delta\ (C \succ\!\!+ i)\ \varphi &= \mathrm{PC}\ \sigma\ \delta\ i\ \varphi \\
\mathrm{PC}\ (\mathrm{READ}\ C)\ \sigma\ \delta\ (C' \succ\!\!+ i)\ (\mathrm{CP}\ \sigma'\ \delta'\ i'\ \varphi) &= \mathrm{PC}\ \sigma'\ \delta'\ i'\ \varphi \\
\mathrm{PC}\ (\mathrm{CALL}\ \sigma\ \sigma')\ \delta\ i\ \varphi &= \mathrm{PC}\ \sigma\ (\mathrm{DF}\ \sigma'\ \delta)\ i\ \varphi \\
\mathrm{PC}\ (\mathrm{RETURN}\ \sigma)\ (\mathrm{DF}\ \sigma'\ \delta)\ i\ \varphi &= \mathrm{PC}\ \sigma'\ \delta\ i\ \varphi \\
\mathrm{PC}\ (\mathrm{TRY}\ \sigma'\ \sigma)\ \delta)\ i\ \varphi &= \mathrm{PC}\ \sigma\ \delta\ i\ (\mathrm{CP}\ \sigma'\ \delta\ i\ \varphi) \\
\mathrm{PC}\ (\mathrm{JUMP}\ \sigma'\ \sigma)\ \delta\ i\ \varphi &= \mathrm{PC}\ \varphi'\ \delta\ i\ \varphi \\
\mathrm{PC}\ (\mathrm{EOS}\ \sigma)\ \delta\ i\ (\mathrm{CP}\ \sigma'\ \delta'\ i'\ \varphi) &= (i = [\,] \rightarrow [\mathit{true}]\ [\!]\ [\,]) \\
&\quad +\!\!+\ (\mathrm{PC}\ \sigma'\ \delta'\ i'\ \varphi) \\
\mathrm{PC}\ (\mathrm{DONE}\ \sigma)\ \delta\ i\ \varphi &= [\,]
\end{aligned}$$

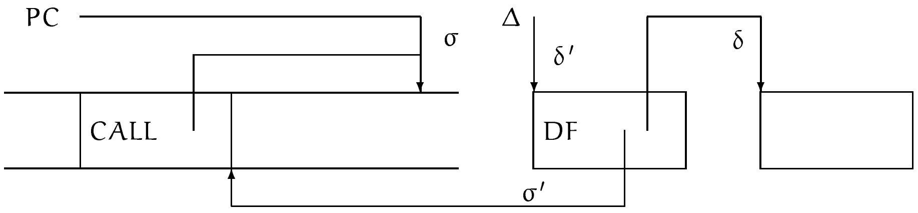
## 5.9.1 Making pointers explicit

In order to merge the dump and the failure continuation we will assume that $\mathit{Dump}$ and $\mathit{Fail}$ are realized by *linked lists* while $\mathit{Success}$ is realized as a linear array of instructions.

Under this interpretation we may visualize PC (CALL $\sigma$ $\sigma'$) $\delta$ by the following picture, where lowercase greek letters are associated with pointers. $\Delta$ denotes the *location* of the second argument of PC.



and PC $\sigma'$ $\delta'$ *where* $\delta' = $ (DF $\sigma$ $\delta$) by the picture:



Hence changing the configuration as sketched in the first picture into that of the second, models the equation:

$$\text{PC (CALL } \sigma\ \sigma') \ \delta\ i\ \varphi \quad = \quad \text{PC } \sigma \text{ (DF } \sigma'\ \delta)\ i\ \varphi$$

An apparent advantage of explicitly introducing pointers is that the semantics can specify that not the complete dump and failure continuations have to be stored in a dump frame, but only pointers to them.

## Update Schemes

The above picture can be formalized by using update schemes. From this description an actual implementation is usually derived with very little effort.

$$\begin{array}{llll} \text{PC}[\,\sigma\,] & \sigma[\,\text{CALL } \sigma'\,]\sigma'' & \Delta[\,\delta\,] & \\ \Rightarrow & & & \\ \text{PC}[\,\sigma'\,] & & \Delta[\,\delta'\,] & \delta'[\,\text{DF } \sigma''\ \delta\,] \end{array}$$

Similarly the other instructions can be described, PC (JUMP $\sigma$ $\sigma'$) = PC $\sigma$ becomes

$$\begin{array}{ll} \text{PC}[\,\sigma\,] & \sigma[\,\text{JUMP } \sigma'\,]\sigma'' \\ \Rightarrow & \\ \text{PC}[\,\sigma'\,] & \end{array}$$

The instruction RETURN returns to the continuation found on top of the dump stack, PC (RETURN $\sigma$) $\delta = \sigma'$ $\delta'$ *where* (DF $\sigma'$ $\delta'$) $= \delta$, pictorially

$$\begin{array}{llll} \text{PC}[\,\sigma\,] & \sigma[\,\text{RETURN}\,]\sigma' & \Delta[\,\delta\,] & \delta[\,\text{DF } \sigma''\ \delta'\,] \\ \Rightarrow & & & \\ \text{PC}[\,\sigma''\,] & & \Delta[\,\delta'\,] & \end{array}$$

Executing $\mathit{PC}\ (\mathit{TRY\_ME\_ELSE}\ \sigma'\ \sigma)\ \delta\ i\ \varphi = \mathit{PC}\ \sigma\ \delta\ i\ \varphi'\ \mathit{where}\ \varphi' = \mathit{CP}\ \sigma'\ \delta\ i\ \varphi$, creates a new a choice point consisting of the next alternative to be tried, a pointer to the top of the dump stack, a pointer to the current choice point and a pointer to the current input position.

$$\begin{array}{llllll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{TRY\_ME\_ELSE}\ \sigma'\,]\sigma'' & \Delta[\,\delta\,] & \mathrm{I}[\,i\,] & \Phi[\,\varphi\,] & \\
\Rightarrow & & & & & \\
\mathrm{PC}[\,\sigma''\,] & & & & \Phi[\,\varphi'\,] & \varphi'[\,\mathrm{CP}\ \sigma'\ \delta\ i\ \varphi\,]
\end{array}$$

An explicit $\mathit{PC}\ (\mathit{FAIL}\ \sigma)\ \delta\ i\ \varphi = \mathit{PC}\ \sigma'\ \delta'\ i'\ \varphi'\ \mathit{where}\ \mathit{CP}\ \sigma'\ \delta'\ i'\ \varphi' = \varphi$, explores the next open alternative by restoring the pointers found in the current choice point.

$$\begin{array}{llllll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{FAIL}\,]\sigma' & & & \Phi[\,\varphi\,] & \phi[\,\mathrm{CP}\ \sigma''\ \delta\ i\ \varphi'\,] \\
\Rightarrow & & & & & \\
\mathrm{PC}[\,\sigma''\,] & & \Delta[\,\delta\,] & \mathrm{I}[\,i\,] & \Phi[\,\varphi'\,] &
\end{array}$$

A successful $\mathit{PC}\ (\mathit{READ}\ C\ \sigma)\ \delta\ [C \mid j]\ \varphi = \mathit{PC}\ \sigma\ \delta\ j\ \varphi$ advances the input pointer over the next character.

$$\begin{array}{llll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{READ}\ C\,]\sigma' & \mathrm{I}[\,i\,] & i[\,C\,]j \\
\Rightarrow & & & \\
\mathrm{PC}[\,\sigma'\,] & & \mathrm{I}[\,j\,] &
\end{array}$$

Otherwise a failure occurs by backtracking to the next open alternative.

$$\begin{array}{lllllll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{READ}\ C\,]\sigma' & & \mathrm{I}[\,i\,] & \Phi[\,\varphi\,] & i[\,C'\,]j & \varphi[\,\mathrm{CP}\ \sigma''\ \delta\ i''\ \varphi'\,] \\
\Rightarrow & & & & & & \\
\mathrm{PC}[\,\sigma''\,] & & \Delta[\,\delta\,] & \mathrm{I}[\,i''\,] & \Phi[\,\varphi'\,] & &
\end{array}$$

### 5.9.2 Merging Dump and Failure

In the above description, no decision has been made on how space for the dump and the failure continuation is allocated. One of the main characteristics of the WAM is that the dump and failure continuation are merged into a single stack called the *local stack*.

The following set of update schemes refines the ones given earlier by allocating frames on the *local* stack, where HP points to the last allocated frame.

The simplest case is CALL

$$\begin{array}{lllll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{CALL}\ \sigma'\,]\sigma'' & \Delta[\,\delta\,] & \mathrm{HP}[\,h\,] & \\
\Rightarrow & & & & \\
\mathrm{PC}[\,\sigma'\,] & & \Delta[\,\delta'\,] & \mathrm{HP}[\,\delta'\,] & \delta'[\,\mathrm{DF}\ \sigma''\ \delta\,]h
\end{array}$$

The RETURN instruction remains unchanged, the dump-frame cannot be reclaimed (but §detret)

$$\begin{array}{llll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{RETURN}\,]\sigma' & \Delta[\,\delta\,] & \delta[\,\mathrm{DF}\ \sigma''\ \delta'\,] \\
\Rightarrow & & & \\
\mathrm{PC}[\,\sigma''\,] & & \Delta[\,\delta'\,] &
\end{array}$$

TRY_ME_ELSE allocates a choicepoint on the local stack.

$$\begin{array}{llllll}
\mathrm{PC}[\,\sigma\,] & \sigma'[\,\mathrm{TRY\_ME\_ELSE}\ \sigma'\,]\sigma'' & \Delta[\,\delta\,] & \mathrm{I}[\,i\,] & \Phi[\,\varphi\,] & \mathrm{HP}[\,h\,] \\
\Rightarrow & & & & & \\
\mathrm{PC}[\,\sigma''\,] & & & & \Phi[\,\varphi'\,] & \mathrm{HP}[\,\varphi'\,] \\
 & & & & & \varphi'[\,\mathrm{CP}\ \sigma'\ \delta\ i\ \varphi\,]h
\end{array}$$

When backtracking the pointer HP can be reset to the position it had when the choice point was created.

$$\begin{array}{lllllll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{FAIL}\,]\sigma' & & & \Phi[\,\varphi\,] & & \varphi[\,\mathrm{CP}\ \sigma''\ \delta\ i\ \varphi'\,]h \\
\Rightarrow & & & & & & \\
\mathrm{PC}[\,\sigma''\,] & & \Delta[\,\delta\,] & \mathrm{I}[\,i\,] & \Phi[\,\varphi'\,] & \mathrm{HP}[\,h\,] &
\end{array}$$

We leave READ and JUMP to the reader.

### 5.9.3 A split stack architecture

A simpler and probably faster machine results when the dump and the failure continuations are kept seperate. The dump must still be realized by means of a linked list but the failure stack can be implemented as a real stack. As a consequence, the pointer to the previous choice point need not be saved. We only give the plans for the relevant instructions.

$$\begin{array}{lllll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{CALL}\ \sigma'\,]\sigma'' & \Delta[\,\delta\,] & \mathrm{HP}[\,h\,] & \\
\Rightarrow & & & & \\
\mathrm{PC}[\,\sigma'\,] & & \Delta[\,\delta'\,] & \mathrm{HP}[\,\delta'\,] & \delta'[\,\mathrm{DF}\ \sigma''\ \delta\,]h
\end{array}$$

An advantage of the split stack architecture is that the dump can be trimmed when setting a choice point.

$$\begin{array}{llllll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{TRY\_ME\_ELSE}\ \sigma'\,]\sigma'' & \Delta[\,\delta\,] & \mathrm{I}[\,i\,] & \Phi[\,\varphi\,] & \\
\Rightarrow & & & & & \\
\mathrm{PC}[\,\sigma''\,] & & & & \Phi[\,\varphi'\,] & \mathrm{HP}[\,\delta\,] \\
 & & & & & \varphi'[\,\mathrm{CP}\ \sigma'\ \delta\ i\,]\varphi
\end{array}$$

Upon failure the computational state is restored with the values found in the topmost choice point.

$$\begin{array}{lllllll}
\mathrm{PC}[\,\sigma\,] & \sigma[\,\mathrm{FAIL}\,]\sigma' & & & \Phi[\,\varphi\,] & & \varphi[\,\mathrm{CP}\ \sigma''\ \delta\ i\,]\varphi' \\
\Rightarrow & & & & & & \\
\mathrm{PC}[\,\sigma''\,] & & \Delta[\,\delta\,] & \mathrm{I}[\,i\,] & \Phi[\,\varphi'\,] & \mathrm{HP}[\,\delta\,] &
\end{array}$$

### 5.9.4 Multiple Choices

Analyzing the behavior of the WAM in case of a chain of choices, (e.g. our running example)

$$S \;=\; (\mathrm{TRY\_ME\_ELSE}\ S' \circ \mathrm{READ}\ a \circ \mathrm{RETURN})\perp$$

$$\begin{aligned}
S' &= (\mathrm{TRY\_ME\_ELSE}\ S'' \circ \mathrm{READ}\ a \circ \mathrm{READ}\ a \circ \mathrm{RETURN})\ \bot \\
S'' &= (\mathrm{TRY\_ME\_ELSE}\ S''' \circ \mathrm{READ}\ a \circ \mathrm{JUMP}\ S)\ \bot \\
S''' &= (\mathrm{READ}\ a \circ \mathrm{READ}\ a \circ \mathrm{JUMP}\ S)\ \bot
\end{aligned}$$

shows that we can discriminate between $S$ which sets up an initial fresh choicepoint, $S'$ and $S''$ which overwrite that very same choicepoint with a another one differing only in the $success$ field, and $S'''$ which ignores the choicepoint.

Now if we redefine FAIL to leave the choicepoint as it is

$$\mathrm{PC}\ (\mathrm{FAIL}\ \sigma)\ \delta\ i\ (\mathrm{CP}\ \sigma'\ \delta'\ i'\ \varphi) \quad = \quad \mathrm{PC}\ \sigma'\ \delta'\ i'\ (\mathrm{CP}\ \sigma'\ \delta'\ i'\ \varphi)$$

we can refine the above choice chain into

$$\begin{aligned}
S &= (\mathrm{TRY\_ME\_ELSE}\ S' \circ \mathrm{READ}\ a \circ \mathrm{RETURN})\ \bot \\
S' &= (\mathrm{RETRY\_ME\_ELSE}\ S'' \circ \mathrm{READ}\ a \circ \mathrm{READ}\ a \circ \mathrm{RETURN})\ \bot \\
S'' &= (\mathrm{RETRY\_ME\_ELSE}\ S''' \circ \mathrm{READ}\ a \circ \mathrm{JUMP}\ S)\ \bot \\
S''' &= (\mathrm{TRUST\_ME\_ELSE\_FAIL} \circ \mathrm{READ}\ a \circ \mathrm{READ}\ a \circ \mathrm{JUMP}\ S)\ \bot
\end{aligned}$$

where TRY_ME_ELSE sets up a choicepoint

$$\mathrm{PC}\ (\mathrm{TRY\_ME\_ELSE}\ \sigma\ \sigma')\ \delta\ i\ \varphi = \mathrm{PC}\ \sigma'\ \delta\ i\ (\mathrm{CP}\ \sigma\ \delta\ i\ \varphi),$$

which is partially updated by subsequent RETRY_ME_ELSE's:

$$\mathrm{PC}\ (\mathrm{RETRY\_ME\_ELSE}\ \sigma\ \sigma')\ \delta\ i\ (\mathrm{CP}\ \sigma''\ \delta'\ i'\ \varphi) = \mathrm{PC}\ \sigma'\ \delta\ i\ (\mathrm{CP}\ \sigma\ \delta'\ i'\ \varphi),$$

until finally the choicepoint is removed by

$$\mathrm{PC}\ (\mathrm{TRUST\_ME\_ELSE\_FAIL}\ \sigma)\ \delta\ i\ (\mathrm{CP}\ \sigma'\ \delta'\ i'\ \varphi) = \mathrm{PC}\ \sigma\ \delta\ i\ \varphi.$$

## 5.9.5 Determinate returns

In the above semantics, dump frames are only reclaimed when creating a choice point or upon backtracking. Using a simple test it is also possible to reclaim stack space upon a RETURN, provided it is determinate. This then would be the same as a RETURN is a conventional imperative language. A return is determinate if it exits an operator body which has no remaining choice-points. This is the case when the dump pointer $\Delta$ is greater than the choice pointer $\Phi$. We don't believe this gains anything except space, in an imperative language the return stack should be heap allocated as well.

This concludes our main discussion on the Warren Abstract Machine, and we now turn to the Recursive Backup Machine.

## 5.10 Recomputing the input

Equation (5.2), shows that that semantics is not single-threading [95] in the input as well as in the success continuation, because they are copied when making a choice.

We redefine $\mathit{failure}$ in order to make the semantics single threading in the input, and this gives rise to several other redefinitions. The semantics can be made single threading in the input if it is not saved when a choice is made, i.e. the input is abstracted from the failure continuation. So we want to transform $\mathit{failure}$ from $\mathbb{B}*$ into a function of type $\mathit{input} \to \mathbb{B}*$. This could be done as before by inventing a number of injective refinements, but we rather pursue a more informal way now. If the invariant $\sigma\ i\ (\sigma'\ i\ \varphi)$ can be maintained, that is to say the topmost choicepoint always has the same input as the current input, then obviously the input need not be saved when making a choice. We then may simply pass the current input as an argument to the failure continuation.

Equation (5.1) shows that the invariant is destroyed when evaluating $\mathrm{READ}\ C\ \sigma\ (C \rightarrowtail i)\ \varphi$. The following derivation shows how this can be fixed.

$$\begin{array}{ll} & \mathrm{READ}\ C\ \sigma\ (C \rightarrowtail i)\ \varphi \\ = & (\mathrm{READ}\ C \gg \mathrm{FAIL})\ \sigma\ (C \rightarrowtail i)\ \varphi \\ = & \mathrm{READ}\ C\ \sigma\ (C \rightarrowtail i)\ (\mathrm{FAIL}\ \sigma\ (C \rightarrowtail i)\ \varphi) \\ = & \sigma\ i\ (\mathrm{FAIL}\ \sigma\ (C \rightarrowtail i)\ \varphi) \\ = & \sigma\ i\ (\mathrm{UNREAD}\ C\ (\mathrm{FAIL}\ \sigma)\ i\ \varphi) \end{array}$$

where $\mathrm{UNREAD}\ C\ \sigma\ i\ \varphi = \sigma\ (C \rightarrowtail i)\ \varphi$. From this derivation we can distillate a *General Principle For Implementing Backtracking*:

> Either the complete state is saved when making a choice, or the implementation *recomputes* some part of the state by letting every action that modifies that part of the state, update the failure continuation such that the modification will be undone upon backtracking.

Based on this principle we define the recomputing input semantics as follows. The semantic domains are:

$$\begin{array}{rcl} \mathit{action} & = & \mathit{success} \to \mathit{success} \\ \mathit{action}^{-1} & = & \mathit{failure} \to \mathit{failure} \\ \mathit{success} & = & \mathit{input} \to \mathit{failure} \to \mathbb{B}* \\ \mathit{failure} & = & \mathit{input} \to \mathbb{B}* \end{array}$$

The semantic operators are given by

$$\begin{array}{rcl} \mathrm{EMPTY}\ \sigma\ i\ \varphi & = & \sigma\ i\ \varphi \\ \mathrm{FAIL}\ \sigma\ i\ \varphi & = & \varphi\ i \end{array}$$

$$
\begin{aligned}
\mathrm{READ}\ C\ \sigma\ (C \rightarrowtail i)\ \varphi &= (C = C') \rightarrow (\sigma\ i\ (\mathrm{READ}^{-1}\ C\ \varphi)) \talloblong \varphi\ (C \rightarrowtail i) \\
\mathrm{CALL}\ e\ \sigma\ i\ \varphi &= e\ \sigma\ i\ \varphi \\
(e \gg f)\ \sigma\ i\ \varphi &= e\ \sigma\ i\ (\lambda i.f\ \sigma\ i\ \varphi) \\
\mathrm{EOS}\ \sigma\ i\ \varphi &= (i = [\,] \rightarrow [true] \talloblong [\,]) +\!\!+\ \varphi\ i \\
\mathrm{READ}^{-1}\ C\ \varphi\ i &= \varphi\ (C \rightarrowtail i)
\end{aligned}
$$

The meaning of a script is then defined as:

$$
\mathcal{M}[\![\textbf{script}\ E]\!]\ i \quad = \quad (\mathcal{E}[\![E]\!] \circ \mathrm{EOS})\bot\ i\ (\lambda i.[\,])
$$

### 5.10.1 Recomputing the success continuation

We now reformulate our semantics such that it becomes recomputing in both $input$ and $success$. According to the principle of implementing backtracking making the semantics recomputing in the success continuation and input, thus $failure = success \rightarrow input \rightarrow \mathbb{B}*$, means that *every* single instruction that does not fail instantly, must update the failure continuation with an inverse operation, since executing an instruction modifies the succes continuation. In order to be able to be able manipulate continuations explicitly, concrete semantics will be used for the success continuation.

$$
\begin{aligned}
Success &::= \mathrm{EMPTY}\ Success \\
&\quad\mid\ \mathrm{FAIL}\ Success \\
&\quad\mid\ \mathrm{READ}\ symbol\ Success \\
&\quad\mid\ (Action \gg Action)\ Success
\end{aligned}
$$

The following machine was first described by Koster [57]

$$
\begin{aligned}
\mathrm{PC}\ (\mathrm{EMPTY}\ \sigma)\ i\ \varphi &= \mathrm{PC}\ \sigma\ i\ (\mathrm{EMPTY}^{-1}\ \varphi) \\
\mathrm{PC}\ (\mathrm{FAIL}\ \sigma)\ i\ \varphi &= \varphi\ (\mathrm{FAIL}\ \sigma)\ i \\
\mathrm{PC}\ (\mathrm{READ}\ C\ \sigma)\ (C \rightarrowtail i)\ \varphi &= \mathrm{PC}\ \sigma\ i\ (\mathrm{READ}^{-1}\ C\ \varphi) \\
\mathrm{PC}\ (\mathrm{READ}\ C\ \sigma)\ i\ \varphi &= \varphi\ (\mathrm{READ}\ C\ \sigma)\ i \\
\mathrm{PC}\ (e \gg f\ \sigma)\ i\ \varphi &= \mathrm{PC}\ (e\ \sigma)\ i\ (e\ {}^{1-}\!\!\gg f\ \varphi) \\
\mathrm{PC}\ (\mathrm{EOS}\ \sigma)\ i\ \varphi &= (i = [\,] \rightarrow [true] \talloblong [\,]) +\!\!+\ (\varphi\ (\mathrm{EOS}\ \sigma)\ i) \\
\\
\mathrm{EMPTY}^{-1}\ \varphi\ \sigma\ i &= \varphi\ (\mathrm{EMPTY}\ \sigma)\ i \\
\mathrm{READ}^{-1}\ C\ \varphi\ \sigma\ i &= \varphi\ (\mathrm{READ}\ C\ \sigma)\ (C \rightarrowtail i) \\
(e\ {}^{1-}\!\!\gg f)\ \varphi\ (e\ \sigma) &= \mathrm{PC}\ (f\ \sigma)\ (e \gg^{-1} f\ \varphi) \\
(e \gg^{-1} f)\ \varphi\ (f\ \sigma) &= \varphi\ (e \gg f\ \sigma)
\end{aligned}
$$

The meaning of a script should not be surprising:

$$
\mathcal{M}[\![\textbf{script}\ E]\!]\ i \quad = \quad \mathrm{PC}\ (\mathcal{E}[\![E]\!]\ (\mathrm{EOS}\ \bot))\ i
$$

### 5.10.2 Advantages and disadvantages of goal stacking versus goal jumping

In the previous section we have argued that goal stacking needs a structure copying implementation of the success continuation, while goal jumping allows for a structure sharing implementation of the success continuation. In [110] the advantages and disadvantages of goal stacking relative to the above goal jumping approach are discussed. Perhaps the most important remark made there is that goal jumping can be viewed as a source level transformation on expressions by putting the constraint that bodies consist of at most two subexpressions. A definition

$$\textbf{call}\ (Q,R,S)$$

is viewed as being transformed into: $\textbf{call}\ (Q,\textbf{call}\ D_0)$ where $D_0 = \textbf{call}(R,\textbf{call}\ D_1)$ where $D_1 = S$.

## 5.11 Conclusions and future work

In this chapter we have shown how various implementations of backtracking on conventional serial machine architectures can be derived systematically from a continuation style denotational semantics of a backtracking language. Using such a transformational approach makes the relationships that exists between various implementations crystal clear. Presenting the implementations in a functional setting has the advantage that it allows one to concentrate on the essentials of implementing backtracking without getting bogged down in too much operational detail (although according to some one gets bogged down in too much *denotational* detail).

The scope of our derivations can be broadened in at least three directions. Firstly by extending the source language. The language used in this paper is quite limited, lacking things such as meta-calls, unification etc. On the other hand we can extend the class of target machines to which the derivation process heads. Currently we have in mind only classical serial machine architectures, but it might be useful to try to derive compilation schemes for logic languages suitable for implementation on parallel architectures.

**Acknowledgements** I would like to thank Maarten Fokkinga, Arie de Bruin and Ross Paterson for their careful and constructive reading of earlier versions [69] of this chapter.

# Bibliography

[1] Aho, Sethi, and Ullman. *Compilers, Techniques and Tools*. Addison-Wesley, 1986.

[2] Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT press, 1991.

[3] Andrew Appel. Heap allocation can be cheaper than stack allocation. *Information Processing Letters*, 25:275–279, 1987.

[4] J. Arsac and Y. Kodratoff. Some techniques for recursion removal. *ACM Toplas*, 4(2):295–322, 1982.

[5] Stijn Arts and Reijer Grimbergen. Taking a risc; implementing functional languages on a risc. Master's thesis, University of Nijmegen, 1990.

[6] Roland Backhouse, Ed Voermans, and Jaap van der Woude. A relational theory of types. In *Proc EURICS Workshop on Calculational Theories of Program Structures*, 1991. email: `wsinrcb@win.tue.nl`.

[7] Hans Bekič. *Programming Languages and Their Definition*. LNCS 177, 1984. Selected Papers edited by C.B. Jones.

[8] Rudolf Berghammer. On the use of composition in transformational programming. Technical Report TUM-I8512, TU München, 1985.

[9] Michel Billaud (`billaud@geocub.greco-prog.fr`). Request for Prolog folk theorems. comp.lang.prolog, May 1991.

[10] Richard Bird. *Programs and Machines: An introduction to the Theory of Computation*. Wiley, 1976.

[11] Richard Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer Verlag, 1987. Also Technical Monograph PRG-56, Oxford University, October 1986.

[12] Richard Bird. Constructive functional programming. In M. Broy, editor, *Marktoberdorf International Summer school on Constructive Methods in Computer Science*, NATO Advanced Science Institute Series. Springer Verlag, 1989.

[13] Richard Bird and Phil Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[14] Eerke Boiten. The many disguises of accumulation. Technical Report 91-26, University of Nijmegen, Department of Computer Science, 1991.

[15] R. Bos and C. Hemerik. An introduction to the category-theoretic solution of recursive domain equations. Technical Report TRCSN 88/15, Eindhoven University of Technology, October 1988.

[16] Manfred Broy. *Transformation parallel ablaufender Programme*. PhD thesis, TU München, München, 1980.

[17] R.M. Burstall and P.J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4, 1969.

[18] M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley, 1988.

[19] L.M. Chirica. *Contributions to Compiler Correctness*. PhD thesis, University of California at LA, USA, 1976.

[20] F.W. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer Verlag, 1981.

[21] P.L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[22] D. Maier and D. Warren. *Computing with Logic*. The Benjamin/Cummings Publishing Company Inc, 1988.

[23] J.W. de Bakker. Recursive procedures. Technical Report Mathematical Centre Tracts 24, Mathematisch Centrum, 1971.

[24] J.W. de Bakker. Comparative semantics for flow of control in logic programming without logic. Technical Report CS-R8840, CWI, October 1988.

[25] A. de Bruin. *Experiments With Continuation Semantics: jumps, backtracking, dynamic networks*. PhD thesis, Vrije Universiteit Amsterdam, 1986.

[26] A. de Bruin and E.P. de Vink. Continuation semantics for Prolog with cut. In J. Diaz and F. Orejas, editors, *LNCS 351: TAPSOFT '89*, pages 178–192. Springer, 1989.

[27] A. de Bruin and E.P. de Vink. Retractions in comparing Prolog semantics. In *Computer Science in the Netherlands 1989*, pages 71–90. SION, 1989.

[28] Peter de Bruin. Naturalness of polymorphism. Technical Report CS 8916, University of Groningen, 1989.

[29] Oege de Moor. Relations in optimization problems. In *Lecture Notes Algoritmics Summerschool Ameland*, 1989.

[30] E.P. de Vink. Comparative semantics for Prolog with cut. Technical Report IR-166, VU, 1988.

[31] Saumya K. Debray and Prateek Mishra. Denotational and operational semantics for prolog. *J. Logic Programming*, 5:61–91, 1988.

[32] Saumya K. Debray and David S. Warren. Functional computations in logic programs. *ACM Toplas*, 11(3):451–481, 1989.

[33] Peter Dybjer. Using domain algebras to prove the correctness of a compiler. In *LNCS 182*. Springer Verlag.

[34] P. Naur (editor). Revised report on the language Algol60. *CACM*, 6(1):1–17, 1963.

[35] Maarten Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4), 1989.

[36] Maarten Fokkinga. Many-sorted algebras. *The Squiggolist*, 2(2), 1991.

[37] Maarten Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, dept of Informatics, Enschede, The Netherlands, 1992.

[38] Maarten Fokkinga, Johan Jeuring, Lambert Meertens, and Erik Meijer. Translating attribute grammars into catamorphisms. *The Squiggolist*, 2(1), 1991.

[39] Maarten Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report 91-4, CWI, 1991.

[40] Nissim Francez, Boris Klebansky, and Amir Pnueli. Backtracking in recursive computations. *Acta Informatica*, 8:125–144, 1977.

[41] H.J. Genrich. Predicate/transition nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986, Part 1*, pages 207–248. Springer Verlag, 1986. LNCS 254.

[42] Françis Giannesini, Henry Kanoui, Robert Pasero, and Michel van Caneghem. *Prolog*. Addison-Wesley, 1986.

[43] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *JACM*, 24(1):68–95, 1977.

[44] Michael J.C. Gordon. *The Denotational Description of Programming Languages*. Springer Verlag, 1979.

[45] R.E. Griswold and M.T. Griswold. *The Icon Programming Language*. Prentice-Hall, 1983.

[46] R.E. Griswold, J. Poage, and H.R. Wiehle. *The SNOBOL4 Programming Language*. Prentice-Hall, 1971.

[47] C. Gunter, P. Mosses, and D. Scott. Semantic domains and denotational semantics. In *Marktoberdorf International Summer school on Logic, Algebra and Computation*, 1989. also in: Handbook of Theoretical Computer Science, North Holland.

[48] Tasuya Hagino. Codatatypes in ML. *Journal of Symbolic Computation*, 8:629–650, 1989.

[49] Wim H. Hesselink. Nondeterminacy and recursion via stacks and games. Technical Report CS 9109, University of Groningen, 1991.

[50] Thomas Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1987.

[51] Neil D. Jones and David A. Schmidt. Compiler generation from denotational semantics. In Neil D. Jones, editor, *LNCS 94: Workshop on Semantics Directed Compiler Generation*. Springer, 1980.

[52] H.B.M. Jonkers. *Abstraction, Specification and Implementation Techniques*. PhD thesis, Mathematisch Centrum, 1982.

[53] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988. Second Edition.

[54] Paul Klint. An overview of the programming language summer. In *ACM POPL 7*, pages 47–55, 1980.

[55] D.E. Knuth. Semantics of context-free languages. *Mathemathical Systems Theory*, 2:127–145, 1968.

[56] C.H.A. Koster. Affix grammars. In J.E.L. Peck, editor, *Algol 68 Implementation*, pages 95–109. North Holland, 1971.

[57] C.H.A. Koster. A technique for parsing ambiguous grammars. In *LNCS 26*. Springer, 1974.

[58] Jean-Louis Krivine. Un interpréteur du λ-calcul. 1985.

[59] Peter Kursawe. How to invent a prolog machine. *New Generation computing*, 5:97–114, 1987.

[60] Peter Lee. *Realistic Compiler Generation*. MIT press, 1990.

[61] D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: a synthetic approach. *Math. Systems Theory*, 14:97–139, 1981.

[62] Joop Leo. On the complexity of top-down recursive backup parsing. Technical Report 80, University of Nijmegen, Department of Computer Science, 1986.

[63] Grant Malcolm. *Algebraic Types and Program Transformation*. PhD thesis, University of Groningen, The Netherlands, 1990.

[64] Ernest G. Manes and Michael A. Arbib. *Algebraic Approaches to Program Semantics*. Springer Verlag, 1986.

[65] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Symposium on Applied Mathematics 19*, 1967.

[66] John McCarthy. Towards a mathematical science of computation. In *Information Processing 1962*. IFIP, North-Holland, 1962.

[67] Lambert Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[68] Lambert Meertens. Paramorphisms. To appear in Formal Aspects of Computing, 1992.

[69] Erik Meijer. A taxonomy of backtracking. In *Computer Science in the Netherlands 1989*. SION, 1989.

[70] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings FPCA'91*. ACM, 1991. LNCS 523.

[71] Erik Meijer and Ross Paterson. Down with lambda-lifting! copies: `erik@cs.kun.nl`, 1991.

[72] Hans Meijer. *Programmar: A Translator Generator*. PhD thesis, University of Nijmegen, Toernooiveld, Nijmegen, The Netherlands, 1986.

[73] John-Jules Ch. Meyer. *Programming calculi based on fixed point transformations: semantics and applications*. PhD thesis, Vrije Universiteit, Amsterdam, 1985.

[74] John-Jules Ch. Meyer. Process modalities obtained as fixed points (parts one and two). *ETACS Bulletin*, 32733, 1987.

[75] R.E. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Wiley, 1976. 2 volumes.

[76] F.L. Morris. Advice on structuring compilers and proving them correct. In *ACM POPL 3*, pages 144–152, 1973.

[77] Peter Mosses. A constructive approach to compiler correctness. In Neil D. Jones, editor, *LNCS 94: Workshop on Semantics Directed Compiler Generation*. Springer, 1980.

[78] Peter Mosses. Abstract semantic algebras! In D. Bjørner, editor, *Formal Description of Programming Concepts II*, pages 63–88. North-Holland, 1983.

[79] Motorola Semiconductors. *m88100 Processor Manual*.

[80] Hugh Osborne. The semantics of update schemes. In *Proceedings of the Code91 Workshop on Code Generation*, 1991. Springer Workshops in Computer Science.

[81] Hugh Osborne. Update plans. In *Proceedings HICSS-25*. IEEE and ACM, 1992.

[82] Ross Paterson. Denotational semantics of term graphs (draft). Imperial College.

[83] Ross Paterson. *Reasoning about Functional Programs*. PhD thesis, University of Queensland, Brisbane, 1988.

[84] Ross Paterson. Operators. In *Lecture Notes Algoritmics Summerschool Ameland*, 1990.

[85] L.C. Paulson. *Logic and Computation*. Cambrige University Press, 1987. Cambridge Tracts in Theoretical Computer Science 2.

[86] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institutes für Instrumentelle Mathematik, Bonn, 1962.

[87] M. Pettersson. Generating efficient code from continuation semantics. In D. Hammer, editor, *Proceedings Third International Workshop on Compiler Compilers*, pages 165–178. Springer, 1990. LNCS 477.

[88] Simon Peyton-Jones and Jon Salkild. The spineless tagless g-machine. In *Proc. 1989 ACM Conference on Lisp and Functional Programming*, 1989.

[89] Gordon Plotkin. Lecture notes on semantics. Edinburgh University.

[90] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator*. Springer Verlag, 1988.

[91] John C. Reynolds. Definitional interpreters for higher order programming languages. In *25th ACM Anual Conference*, pages 717–740, 1972.

[92] John C. Reynolds. On the relation between direct and continuation semantics. In *2nd Colloqium on Automata and Languages*, pages 141–157, 1974. LNCS 14.

[93] John C. Reynolds. Types abstraction and parametric polymorphism. In *Information Processing '83*. North Holland, 1983.

[94] R. Salter and J. Jones. Graph reduction models of logic programming control. NSF proposal CCR-9002132.

[95] David A. Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[96] D.S. Scott. A type-theoretic alternative to cuch, iswim, owhy. Typed notes, Oxford, 1969.

[97] Peter Sestoft and Harald Søndergaard. A bibliography on partial evaluation. *SIGPLAN Notices*, 23(2), 1988.

[98] Ravi Sethi. Circular expressions: Elimination of static environments. *Science of Computer Programming*, 1:203–222, 1982.

[99] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–785, November 1982.

[100] Joseph E. Stoy. *Denotational Semantics, The Scott-Strachey Approach to Programming Language Theory*. The MIT press, 1977.

[101] J.W. Tatcher, E.C Wagner, and J.B Wright. More advice on structuring compilers and proving them correct. In Neil D. Jones, editor, *LNCS 94: Workshop on Semantics Directed Compiler Generation*. Springer, 1980.

[102] R.D. Tennent. The denotational semantics of programming languages. *CACM*, 19(8):437–453, 1976.

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

[104] Nico Verwer. Homomorphisms, factorisation and promotion. *The Squiggolist*, 1(3), 1990. Also technical report RUU-CS-90-5, Utrecht University, 1990.

[105] Phil Wadler. Views: A way for pattern matching to cohabit with data abstraction. Technical Report 34, Programming Methodology Group, University of Göteborg and Chalmers University of Technology, March 1987.

[106] Phil Wadler. Theorems for free ! In *Proc. 1989 ACM Conference on Lisp and Functional Programming*, pages 347–359, 1989.

[107] Phil Wadler. Comprehending monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, 1990.

[108] M. Wand. Fixed point constructions in order enriched categories. *Theoretical Computer Science*, 8, 1979.

[109] M. Wand. A short proof of the lexical addressing algorithm. *Information Processing Letters*, 35:1–5, 1990.

[110] D. Warren. An abstract prolog instruction set. Technical Report 309, SRI, 1983.

[111] D.A. Watt. Executable semantic descriptions. *Software Practice and Experience*, 16(1):13–43, 1986.

[112] D.A. Watt and O.L. Madsen. Extended attribute grammars. *The Computer Journal*, 26(2):142–153, 1983.

[113] G.C. Wraight. Categorical datatypes — a critique of hagino's thesis. November 1988.

[114] Hans Zierer. Programmierung mit funktionsobjecten: Konstruktive erzeugung semantische bereiche und anwendung auf die partielle auswertung. Technical Report TUM-I8803, TU München, 1988.

# Samenvatting

Er is sprake van een zekere mate aan discrepantie tussen de theorie van de semantiek van programmeertalen en de praktijk van de vertalerbouw. Dit proefschrift poogt een brug te slaan tussen deze twee werelden. Uitgaande van een formele semantiek wordt in een aantal stappen een concrete implementatie berekend. In een formele semantiek wil je alleen uitdrukken *wat* een programma betekent, bijvoorbeeld als functie van invoer naar uitvoer. Een concrete implementatie moet juist precies aangeven *hoe* de berekening van de output bij gegeven input plaats vindt. Een verfijningsstap van een abstracte naar een meer operationele semantiek bestaat meestal uit het zichtbaar maken van een impliciete evaluatie volgorde door het introduceren van een continuatie. Een continuatie is een expliciete vorm van een berekening die 'later' nog gedaan moet worden. Als alle noodzakelijke evaluatie volgorde is vastgelegd kunnen er concrete realizaties worden gekozen voor continuaties en andere semantische domeinen; stacks, verketende lijsten, machine code, registers etc. We leiden vertalers af voor een eenvoudige functionele, een imperatieve en een logische programmeertaal.

Voor het beschrijven van laagbijdegrondse operaties van abstracte machines gebruiken we de *update schemes* van Hans Meijer. Het voordeel van update schemes is dat complexe bewerkingen op pointers en arrays op abstracte wijze kunnen worden gedefinieerd. Ook is het relatief eenvoudig een update scheme specificatie te implementeren in een concrete taal als C. Een nadeel is dat update schemes zich minder goed lenen voor verificatie en transformatie.

Als formalisme voor het definieren en transformeren van semantische functies gebruiken we een variant van *Squiggol*, een calculus voor functies op algebraische datatypes ontwikkeld door Richard Bird en Lambert Meertens. We mogen stellen dat het gebruik van Squiggol een succes is voor wat betreft het werken met algebraische datatypes. De aanwezigheid van functieruimtes gooit echter behoorlijk wat roet in het eten, een verschijnsel dat zich, gelukkig, bij anderen ook voordoet. Het rekenen met functieruimtes is dan ook een van de belangrijkste thema's voor verder onderzoek. Andere zaken die nadere aandacht verdienen zijn mogelijke verbanden met partiele evaluatie , in het bijzonder het bepalen van de grens tussen statische en dynamische berekeningen (bindig time analysis) en machine assistentie. Wil de hier gepropageerde methode ook praktische waarde hebben is mechanische hulp onontbeerlijk.

# Curriculum Vitae

**1963** Geboren te Willemstad, Curaçao N.A. op 18 april.

**1981-1986** Studie Informatica aan KUN (cum laude).

**1986-1992** Promovendus 'oude stijl' bij de vakgroep Informatica van de KUN.

**1992-** Postdoc onderzoeker bij de afdeling Informatica van de RUU (`erik@cs.ruu.nl`).