Sharing of Computations

Torben Amtoft
Computer Science Department
Aarhus University, Ny Munkegade, DK-8000 Århus C,
Denmark
`internet: tamtoft@daimi.aau.dk`

August 29, 1993

Danish summary

Denne rapport er en revideret udgave af min afhandling af samme navn, som i juni 1993 blev accepteret til forsvar af PhD-graden i datalogi ved Aarhus Universitet.

Motivation

I de senere år har man arbejdet meget med at udvikle værktøjer til at gøre programmer mere effektive. Af teknikker kan nævnes memoisering [Kho90]; udfold/fold transformationer [PP91b]; graf-baseret implementation af "lazy" evaluering [Jon87] og partiel evaluering [JSS89]. At disse metoder forbedrer effektiviteten skyldes at nogle beregninger *deles*, så de kun skal gøres én gang. Imidlertid er forbindelsen mellem teknikkerne ikke klart forstået, og det er heller ikke klart hvor stor effektivitetsforbedring (*speedup*) de kan forårsage. Ydermere giver anvendelse af teknikkerne udfold/fold og partiel evaluering risiko for ødelæggelse af termineringsegenskaber.

Den existerende litteratur inden for området vidner om mangel på en model for program udførelse/transformering der er abstrakt nok. Behandlingen har været for afhængig af det konkrete sprog/system, og derfor er de essentielle begreber ofte druknet i detaljer.

Formålet med denne afhandling er at præsentere en model (faktisk to, nemlig én for et funktionelt sprog og én for et logisk sprog) som jeg tror/håber vil hjælpe til med at isolere de karakteristiske træk ved optimeringsteknikker der er baserede på at beregninger deles.

Modellen er basered på en transitionssemantik i Plotkin-stil [Plo81]. Grunden til at en transitionssemantik foretrækkes frem for en denotationel semantik er at førstnævnte bedre fanger at udfoldning/foldning er *operationelle* begreber.

Hovedidéen, at bruge *mange-niveau transitionssystemer*, er som følger:

- Det oprindelige program (kildeprogrammet) er repræsenteret som *regler på niveau 0.*
- At udføre kildeprogrammet modelleres af (en sekvens af) *transitioner på niveau 1*, hvor man på niveau 1 "har adgang til" regler på niveau 0.

- At transformere kildeprogrammet (at foretage "symbolsk evaluering") bliver ligeledes modelleret af transitioner på niveau 1. Resultatet af transformationen, målprogrammet, vil blive repræsenteret som *regler på niveau 1.*

- At udføre målprogrammet modelleres af transitioner på niveau 2, hvor man på niveau 2 har adgang til regler på niveau 1.

At beregninger deles afspejles i at en regel på niveau 1, så snart den er udledt, kan bruges *mange* gange på niveau 2 – hver anvendelse repræsenterer en *genvej* i beregningsprocessen.

Man kan tænke på kildeprogrammet som en samling axiomer i en teori T_0; så kan man opfatte målprogrammet som enten en samling teoremer i T_0 eller som en samling axiomer i en ny teori T_1, som man kan forvente vil være mere effektiv end T_0 – cf. [Gru87].

I vores model kan *korrekthed* (løst sagt) udtrykkes som følger: hvis der er en transition på niveau 2 fra en konfiguration C til en anden konfiguration C', skal der også være en transition på niveau 1 fra C til (noget "ækvivalent med") C' (dette er partiel korrekthed); og hvis der fra en konfiguration C udgår en uendelig kæde af transitioner på niveau 2, skal der også udgå en uendelig kæde af transitioner fra C på niveau 1 (dette er total korrekthed).

Ligeledes kan man ræsonnere om *speedup* (hvis man tildeler hver transition en "omkostning"). F.ex. vil egenskaben at man højest vinder en konstant faktor ved at foretage en given transformation (løst sagt) kunne udtrykkes som følger: der existerer en konstant k således at det (for alle A, B) gælder at hver gang der er en niveau 2 transition fra A til B med omkostning c, findes der også en niveau 1 transition fra A til (noget "ækvivalent med") B med omkostning $\leq kc$.

Bemærk at modellen ser "standard evaluering" som et specialtilfælde af "symbolsk evaluering" (som det også gøres i [DP88], og som det gøres i PROLOG verdenen). På den anden side har man som oftest "lov til" at gøre mere under symbolsk evaluering end under standard evaluering, og dette må tages med i modellen der ellers vil blive både temmelig triviel og med særdeles begrænset anvendelsesområde. Når man f.ex. arbejder med et "lazy" sprog har man under standard evaluering kun lov til at reducere det "yderste redex", mens man typisk har lov til at reducere et vilkårligt redex under symbolsk evaluering; og når man arbejder med PROLOG har man under standard evaluering kun lov til at kalde prædikatet yderst til venstre, mens man ofte har lov til at kalde et vilkårligt prædikat under symbolsk evaluering.

I modelskitsen præsenteret ovenfor var kun 2 niveauer angivet, men man kan naturligvis generalisere til n niveauer – og det er også nødvendigt hvis man skal modellere memoisering hvor man udnytter tidligere genererede regler til at generere nye regler, som f.ex. når fibonacci programmet via memoisering kører i lineær tid i stedet for i exponentiel tid.

En vigtig begrebsmæssig forskel mellem det ovenfor angivne perspektiv på transformationer og det perspektiv som er fremherskende i litteraturen (f.ex. [BD77]) er som følger:

- i den klassiske begrebsramme bliver kildeprogrammet, via en sekvens af meningsbevarende trin, transformeret ind i målprogrammet;
- i vores begrebsramme "observerer" man hvordan kildeprogrammet opfører sig, og ved hjælp af den information konstruerer man så et målprogram.

For en nærmere sammenligning af de to perspektiver se f.ex. [Tur86, p. 293], ifølge hvilken førstnævnte er "suggested by axiomatic mathematics" og sidstnævnte er "a product of cybernetic thinking".

En oversigt over afhandlingen

- I kapitel 2 uddyber vi ovennævnte behandling af mange-niveau transitionssystemer. Vi kigger på flere velkendte teknikker for optimering af programmer, diskuterer deres fordele og begrænsninger og viser hvordan de passer ind i vores begrebsramme.
- Kapitel 3 foregår i den funktionelle verden; vi betragter evalueringsstrategier for λ-kalkylen så vel som for "supercombinator" programmer. Det er velkendt at "lazy" evaluering er suboptimal mht. evnen til at genbruge beregninger; det er mindre kendt at også "fully lazy" evaluering er suboptimal – kan endda være exponentielt dårlig, som vist i [FS91].

  For evaluering af λ-udtryk existerer der adskillige smarte metoder som genbruger beregninger i højere grad end "fully lazy" evaluering gør (f.ex. [Lam90]). I sektion 3.1 præsenterer vi en *parametriseret* evalueringsstrategi for supercombinator programmer med det fordringsfulde navn “ultimate sharing”. Denne strategi er en “top-down” implementering af et mange-niveau transitionssystem; hvis parametrene bliver valgt på passende vis er strategien i stand til at opnå samme genbrug af beregninger som de ovenfor nævnte metoder.

Kapitlet søger at forene og klargøre idéer fra forskellige steder i litteraturen, deriblandt [AT89] (Jesper Träff's og mit speciale).

- Kapitel 4 kan nok betragtes som hovedkapitlet i afhandlingen; et kort sammendrag følger nedenfor (kapitlet selv indledes med et mere detaljeret sammendrag):

  1. I sektion 4.1-4.5 bliver begrebet mange-niveau transitionssystemer formaliseret i en funktionel ramme, hvor konfigurationerne er grafer. Det vises at niveau 1 transitioner (dvs. standard evaluering) tilfredsstiller en Church-Rosser egenskab, og at “normal order reduction” (svarende til “lazy” evaluering) er en optimal strategi (i forhold til andre strategier på niveau 1).

     Så længe kun ét niveau er til stede, har lignende fremgangsmåder været anvendt adskillige andre steder i litteraturen – imidlertid giver vi behandlingen en drejning så den passer til vores senere formål.

  2. Sektion 4.6 behandler de centrale emner *korrekthed* og *speedup*. Sætning 4.6.3 kan fortolkes som udsigende at målprogrammet højest er en konstant faktor hurtigere end kildeprogrammet; og sætning 4.6.4 kan fortolkes som udsigende at såfremt antallet af niveauer er opadtil begrænset er højest et *polynomielt* speedup muligt. Endvidere opstiller sætning 4.6.7 en tilstrækkelig betingelse for *total korrekthed* – fidusen er at sikre at hver regel repræsenterer en smule “fremgang”.

     Sektion 4.7 indeholder en detaljeret diskussion af hvordan og hvorvidt de ovennævnte resultater kan anvendes på den “virkelige verden”, f.ex. de mange-niveau transitionssystemer der blev behandlet i kapitel 2.

     I sektion 4.8 bliver nogle ingredienser “faktoriseret ud”, som hver for sig kan give *mere end* et konstant (polynomielt) speedup.

Dvs. vi undersøger de antagelser der ligger bag sætning 4.6.3 og sætning 4.6.4.

De fleste af de idéer, som bliver fremstillet i disse tre sektioner, blev præsenteret til PEPM'91 [Amt91] (da blev imidlertid en model for logiksprog brugt).

Sektion 4.9 sætter arbejdet i perspektiv ved at beskrive andre fremgangsmåder fra litteraturen.

- I kapitel 5 defineres en non-deterministisk maskine (kaldet en USM) der implementerer "ultimate sharing" (cf. kapitel 3). Ved at anvende resultater fra kapitel 4 kan det vises at maskinen er "korrekt".

  En USM kan gøres deterministisk på mange måder; af særlig interesse er den såkaldte PEM som behandles i sektion 5.2. En PEM er essentielt en "top-down" implementering af partiel evaluering.

  Materialet i dette kapitel kan ses som en generalisering af hovedidéen bag Jesper Träff's og mit speciale [AT89].

- Kapitel 6 giver et exempel på et "realistisk" program som ved hjælp af en passende instans af en USM kan komme til at køre exponentielt hurtigere. Det drejer sig om en simulator for de såkaldte *2DPDA*'s [AHU74, kapitel 9]. Det vakte stor opsigt da Cook beviste at det altid er muligt at simulere en 2DPDA i lineær tid, selv hvis automaten udfører exponentielt mange trin. Vi skal se at denne smarte simulering kan ses som en instans af det generelle begreb "ultimate sharing".

  Kapitlet er baseret på *fælles arbejde med Jesper Träff*, som er publiceret i TCS [AT92] (den grundlæggende idé går tilbage til specialet [AT89]). Fremstillingen her vil være væsentligt anderledes, da vi kan udnytte den generelle teori udviklet i de foregående kapitler.

- Også i kapitel 7 ser vi et exempel på hvordan "smarte algoritmer" kan genopfindes via anvendelse af generelle teknikker for programoptimering. Vi viser at de velkendte Knuth-Morris-Pratt (KMP) og Boyer-Moore (BM) algoritmer til delstrengsgenkendelse kan ses som instanser af en *fælles* algoritme, *parametriseret* mht. søgestrategi. Som sidegevinst bliver det således formaliseret at KMP og BM er "duale".

- I kapitel 8 præsenteres en model for et *logisk sprog*. Hovedvægten lægges på at give tilstrækkelige betingelser for total korrekthed af udfold/fold transformationer.

  De grundlæggende idéer i kapitlet blev præsenteret til PLILP'92 [Amt92a]; bortset fra de indledende dele er kapitlet næsten identisk med den tekniske rapport [Amt92b].

- Kapitel 9 søger at træde et skridt tilbage og betragte hvad der er opnået. I særdeleshed findes to emner værdige til en nærmere diskussion:

  - i og med at det er essentielt for anvendelser af "ultimate sharing" at vores USM får velvalgte parametre, er det en påtrængende opgave at udvikle analyseværktøjer der kan hjælpe brugeren med dette;
  - da motivationen var at lave en model der er uafhængig af konkrete programmeringssprog, virker det utilfredsstillende at afhandlingen introducerer *to* (relativt forskellige) modeller – kan "en større fælles kerne" for den funktionelle model og den logiske model findes?

Contents

Preface to the revised version

This report is a revised version of my thesis of the same title, which was accepted for the Ph.D. degree in Computer Science at University of Aarhus, Denmark, in June 1993.

The examiners (Neil Jones and Alberto Pettorossi) made many useful remarks, which helped me to see how to improve the original thesis. In particular, I realized that section 8.1 (where the logic model is outlined) could benefit from a major rewriting. Apart from that, only minor changes have been made.

Chapter 1

Introduction

In the recent years a lot of work has been devoted to developing tools for transforming less efficient programs into more efficient programs. These include memoization [Kho90]; unfold/fold transformations [PP91b]; graph-based implementation of lazy evaluation [Jon87] and partial evaluation [JSS89]. The efficiency improvement caused by these techniques all are due to the fact that some computations are shared, i.e. they only have to be done once. However, it is in no way clearly understood how these techniques relate to each other; neither is it clearly understood how much speedup one can gain. Finally, it is no easy task to guarantee preservation of termination properties for the techniques of unfold/fold and partial evaluation.

From the literature on the techniques above one clearly feels the lack of a model for program execution/transformation which is abstract enough. The treatment has been too language-dependent, and accordingly the essential concepts have been overshadowed by details.

The goal of this thesis is to present a model (actually two, namely one for a functional language and one for a logic language) which I believe will help to isolate the characteristic features of program optimization techniques which are based on sharing of computations. The model is based on transitions (defined in Plotkin-style [Plo81]) between configurations; the reason for preferring a transition semantics to a denotational semantics is that the former more naturally expresses the fact that unfolding/folding etc. is operational in nature[1].

The main idea – to use *multilevel transition systems* – is as follows:

- The original program (the source program) is represented as *rules*

[1]Cf. the remark in [GLT89, p. 54]: "The fundamental idea of denotational semantics is to interpret reduction (a dynamic notion) by equality (a static notion)".

at level 0.

- To execute the source program is modeled by (a sequence of) *transitions at level 1*, where one at level 1 "can access" rules at level 0.
- To transform the source program (to do "symbolic evaluation") likewise is modeled by transitions at level 1. The result of transformation (the target program) will then be represented as *rules at level 1.*
- To execute the target program is modeled by transitions at level 2, where one at level 2 can access rules at level 1.

The sharing aspect of the above comes from that fact that a level 1 rule, once derived, can be used *many* times at level 2 – each application representing a *shortcut* in the computation process.

If one thinks of the source program as a set of axioms in a theory T_0, one can consider the target program as either a set of theorems in T_0 or as a set of axioms in a new theory T_1, which one can expect to be more efficient than T_0 – cf. [Gru87].

Within the model one can (loosely speaking) express *correctness* as follows: If there is a transition at level 2 from some configuration C into another configuration C', then there must also be a transition at level 1 from C to (something "equivalent to") C' (partial correctness), and if there from configuration C is an infinite chain of transitions at level 2, there also must be an infinite chain of transitions from C at level 1 (total correctness).

Likewise one can reason about *speedup* if one assigns a "cost" to each transition; e.g. the property that one gains at most a constant by doing some transformation can be expressed as follows: there exists a constant k such that (for all C,C') each time there is a level 2 transition from C to C' with cost c, there is a level 1 transition from C to (something "equivalent to") C' with cost c', where $c' \leq kc$.

Notice that a key point in the above model sketch is that "standard" evaluation is viewed as a special case of "symbolic" evaluation (as done in [DP88], and as done in the PROLOG world). On the other hand, in practice one is allowed to do more during symbolic evaluation than during standard evaluation, and the model has to account for these differences (without doing so the theory will become rather trivial as well as of limited use for modeling purposes). For example, during standard evaluation of a lazy language one will only accept that the outermost redex is reduced, whereas one maybe is entitled to reduce an arbitrary redex during symbolic evaluation. Likewise, in a logic language like PROLOG one is typically only allowed to call the leftmost predicate during standard evaluation, whereas one is allowed to call an arbitrary predicate during symbolic evaluation.

In the model sketched above only 2 levels are present, but of course one can generalize to n levels – and such a generalization is also necessary for modeling memoization where one during generation of rules exploits previously generated rules, as when e.g. the fibonacci program due to memoization runs in linear time instead of in exponential time.

An important conceptual difference between the perspective on transformations presented above and the perspective prevalent in the literature (e.g. [BD77]) is as follows:

- in the standard framework, the source program gradually – by a sequence of (hopefully) meaning preserving steps – is transformed into the target program;
- in our framework the behavior of the source program is "observed", and by means of the information thus gained a target program is constructed.

For a further discussion of the merits of the two perspectives see [Tur86, p. 293], according to which the former is "suggested by axiomatic mathematics" and the latter is "a product of cybernetic thinking".

1.1 An overview of the thesis

- In chapter 2 we elaborate on the concept of multilevel transition systems. In particular we examine several well-known program optimization techniques, discuss their merits and limitations and show how they fit into the framework of multilevel transition systems.
- Chapter 3 takes place within a functional setting, considering evaluation strategies for the λ-calculus as well as for supercombinator programs. It is well-known that lazy evaluation is suboptimal wrt. the ability for reusing (sharing) computations; it is less known that

also “fully lazy” evaluation is suboptimal – it may even be exponentially bad, as shown in [FS91].

Several clever methods exist for “a more than fully lazy” evaluation of λ-expressions (e.g. [Lam90]); in section 3.1 we present a *parametrized* evaluation strategy for supercombinator programs with the pretentious name “ultimate sharing”, to be seen as a “top-down” implementation of a multilevel transition system – we argue that if parameters are chosen appropriately, this strategy is able to achieve the same degree of sharing as the abovementioned methods.

This chapter attempts to unify and elucidate ideas from various places in the literature, including [AT89] (Jesper Träff’s and mine Master’s Thesis).

- Chapter 4 may be considered the main chapter of the thesis, and can be summarized as follows (the chapter itself contains a more detailed overview):

  1. In section 4.1-4.5, the idea of multilevel transition systems is formalized in a functional setting where the configurations are graphs. It is shown that level 1 transitions (i.e. standard evaluation) satisfy a Church-Rosser property, and that “normal order reduction” is optimal among evaluation strategies at level 1.

     As long as only one level is present, developments rather similar to the one presented here have been seen numerous places in the literature – however, our development has been given a twist in order to suit our later purposes.

  2. In section 4.6, the crucial issues of *correctness* and *speedup* are addressed. In particular, theorem 4.6.3 can be interpreted as saying that the target program will be faster than the source program by at most a constant; and theorem 4.6.4 can be interpreted as saying that by having an upper bound on the number of levels employed in a multilevel system one at most gains a *polynomial* speedup. Moreover, theorem 4.6.7 gives criteria for total correctness – the trick is to ensure that each rule represents some “progress”.

     In section 4.7 it is discussed in detail how the above results relate to the “real world”, e.g. the multilevel systems examined in chapter 2.

In section 4.8 we factor out some reasons why a program optimization technique may yield *more than* a constant (polynomial) speedup (that is, we investigate the underlying assumptions behind theorem 4.6.3 and theorem 4.6.4).

The main ideas exposed in these three sections were presented at PEPM'91 [Amt91], however in a logic programming setting.

Finally, section 4.9 attempts to put the present work in perspective by describing other approaches from the literature.

- In chapter 5 an abstract, non-deterministic machine (to be called an USM) implementing ultimate sharing (cf. chapter 3) is defined. By applying results from chapter 4, the machine can be proven "correct".

  Of the numerous instances of the USM, the so-called PEM is of special interest and will be treated in depth in section 5.2. The PEM can be considered a top-down implementation of partial evaluation.

  The material presented in this chapter may be viewed as a generalization of the main idea behind [AT89] (Jesper Träff's and mine Master's Thesis).

- Chapter 6 presents a "realistic" program which by means of a suitable instance of the USM can be made to run exponentially faster. The program to be considered is a simulator for *two-way deterministic pushdown automata* (2DPDA) [AHU74, chap. 9]. It caused much surprise when Cook showed that it is always possible to simulate a 2DPDA in linear time (wrt. the length of the input tape), even if the automaton carries out an exponential number of steps. We shall see that the effect of this clever simulation can be acquired using the general concept of ultimate sharing.

  This chapter is based on joint work with Jesper Träff which has been reported in TCS [AT92] (but the basic idea dates back to [AT89]). The exposition here will be rather different, as we can build upon the general theory developed in the previous chapters.

- As in chapter 6, also in chapter 7 we shall see that ingenious algorithms can be reinvented by application of general program optimization techniques: we show that the well-known Knuth-Morris-

Pratt (KMP) and Boyer-Moore (BM) algorithms for substring matching can be seen as specializations of a *common* substring matching algorithm, *parametrized* wrt. search strategy. This further formalizes the intuition that KMP and BM are "dual".

- In chapter 8 a model is set up for a logic language; special emphasis is put on giving criteria for *total correctness* of unfold/fold transformations.

  The basic ideas in this chapter have been presented at PLILP'92 [Amt92a]; the chapter itself is (apart from the introductory parts) almost identical to the technical report [Amt92b]. Section 8.1 attempts to give the main intuition behind the approach and section 8.2 compares with related work; the rest of the chapter is highly technical and perhaps ought to be an appendix instead.

- Chapter 9 contains the concluding remarks. In particular, two issues are discussed:

  – It is essential for the success of the USM that it is instantiated by appropriate parameters. Can some analysis guide the user?

  – Can the functional model and the logic model be brought closer together?

1.2 Acknowledgements

My academic career can be divided into two rather distinct parts: the graduate study at DIKU, University of Copenhagen and the PhD study at DAIMI, University of Aarhus.

I was initiated to the brave new world of "semantics based program manipulation" in fall 1985, at which time Valentin Turchin was visiting DIKU invited by Neil Jones. Together with Anders Bondorf I struggled to get hold of the basic concepts within the area, and together with Anders Bondorf I made a project guided by Valentin Turchin (and later Neil Jones). More than seven years after, my intuition about program manipulation is still heavily influenced by Valentin Turchin.

To be a graduate student at DIKU was a very special period of life (which I, also for other reasons, tried to prolong as far as possible...), due to the extremely stimulating research environment created by Neil

Jones. Warm thanks to Neil Jones for being a very inspiring advisor, and for patiently providing constructive criticism on numerous drafts making it possible to convert them into readable papers.

Many other members of Neil Jones' group would deserve a mention (e.g. Torben Mogensen for always being able and willing to explain the behavior of even the most complicated program analysis, Olivier Danvy for his infecting enthusiasm, Carsten Kehler Holst for being the one to ask in order to understand what partial evaluation really is about) but in particular I wish to thank Jesper Träff, my partner on Master's Thesis work with whom I have stayed in at least weekly but often daily contact ever since, thanks to `e-mail`.

Life in Århus, Jutland, has been a very pleasing experience – so far no attempts at ethnic cleansing have been made, and I even (from Oct 89 to Sep 92) enjoyed a scholarship from University of Aarhus.

Currently I am supported by the DART-project; thanks to my employers Hanne Riis Nielson and Flemming Nielson for giving me time to complete this thesis. Also thanks to Hanne and Flemming for being the most consistent representatives for the (implicit) DAIMI paradigm: do not have confidence in anything unless you have *proved* it (the DIKU paradigm is not to have confidence in anything unless you have seen that it *runs*).

Thanks to my PhD-advisor Brian Mayoh for always being optimistic, for his broad knowledge within numerous fields of computer science and for his ability to quickly capture the essence of a text.

A very special thank is due to Jens Palsberg for giving valuable comments on a draft of this thesis, and for many fruitful discussions during the last three years – in which period our interests have converged.

Many other people at DAIMI deserve a mention – like Henrik Andersen (now in Copenhagen) for being a most stimulating office mate and Michael Schwartzbach for emphasizing the beauty of keeping things as simple as possible – but in particular I wish to thank Sten Agerholm, my current office mate, for making a three-months visit to Cambridge at the end of 1992 thus giving me single access to a work station and removing my (closest) access to small talk (without this contribution, the present work would not have been possible...).

Finally, I wish to thank my friends outside DAIMI and my family – not least my father, Henning Hansen, who (in spite of being single) has been an outstanding support and encouragement.

Chapter 2

Multilevel transition systems

In this chapter we will elaborate a bit on the intuition presented p. 12 – in particular, we will show how several well-known program optimization techniques can be considered as instances of multilevel transition systems.

A *multilevel system*, in its most bare form, consists of

- A set of *configurations*, C.
- For each natural number i a set of *transitions* T_i, where each T_i is a binary relation on C. Thus $(C_1, C_2) \in T_i$ is supposed to mean that there is a transition from C_1 to C_2 at level i – we will also write $i \vdash C_1 \rightarrow C_2$.
- For each non-negative number i a set of *rules* $\mathcal{R}_i$, where each $\mathcal{R}_i$ is a binary relation on C. We will demand that for all i $\mathcal{R}_i \subseteq T_i$, i.e. that a level i rule is also a level i transition.
- An inference system for the $i \vdash$ relation, where the inference rules are of form

  $$\frac{\ldots i \vdash C_{11} \rightarrow C_{12}, i \vdash C_{21} \rightarrow C_{22} \ldots}{\ldots i \vdash C_1 \rightarrow C_2}$$

  and where there is an axiom stating that

  $$\frac{(C_1, C_2) \in \mathcal{R}_{i'}, i' < i}{i \vdash C_1 \rightarrow C_2}$$

  i.e. at level 1 only the level 0-rules can be used, at level 2 also the level 1-rules are applicable etc. There may be some side conditions; a possible one being that $i = i' + 1$ (so at level 2 one can use the level 1-rules but *not* the level 0-rules).

We say that a multilevel system is an *n-level system* if $\mathcal{R}_i = \emptyset$ for $i \geq n$ (but $\mathcal{R}_{n-1} \neq \emptyset$) – in particular, in a 2-level system only $\mathcal{R}_1$ (and $\mathcal{R}_0$) will be $\neq \emptyset$.

As can be seen, a multilevel system will be determined (given a fixed set of configurations and inference rules) from the sets $\mathcal{R}_i$, since then the sets T_i will be fixed.

Pragmatics

In order to specify and implement a multilevel system a number of issues, some of which are listed below, must be settled. These decisions will then implicitly define the sets $\mathcal{R}_i$.

- Given that a rule at level i with "source" s is wanted, it remains to find the corresponding "target", i.e. a t such that $i \vdash s \to t$. Many such t may exist, representing various degrees of "reduction". Often one will reduce s until some sort of "normal form" is reached; of course care has to be taken to ensure termination.

- Often there is a choice between whether to use the rules in $\mathcal{R}_0$, which can be assumed to be easily accessible, but which represent small computation steps only; or to use the higher level rules, which might be costly to find (and compute), but potentially represent larger computation steps.

- For any $i \geq 1$, it must be settled *which* configurations will be sources of transitions in $\mathcal{R}_i$, and *when* these rules are to be generated. Roughly speaking, there are two ways to proceed:

  – To compute the rules *bottom-up*, i.e. start to compute all the rules wanted as members of $\mathcal{R}_1$, then (if any) the rules in $\mathcal{R}_2$, etc. When all rules are stored, the system is able to evaluate expressions (in the world of logic programming: solve queries), now working at the top level. This approach is thus a *two-stage* technique; the advantage being that the rules can be "compiled" (into more efficient representations) between the two stages – the disadvantage being that rules that are never needed might be generated.

  – To use a *top-down* (or call-by-need) approach: rules are generated only when needed to solve a given query (in an efficient way). This approach is thus a *one-stage* technique; the advantage being that one does not have to determine in advance which rules to generate (without knowing the actual query) – the disadvantage being that a lot of administration overhead is potentially present.

The discussion above will be concretized in the next section.

2.1 Instances of multilevel systems

We now examine three well-known techniques for program optimization and show how their behavior can be expressed in terms of multilevel systems.

2.1.1 Memoization

This is a classical technique, introduced by [Mic68]. The rules are of form

$$f(\alpha) \rightarrow \beta$$

(α and β are constants) making it possible to share the computation of $f(\alpha)$ between its various invocations. When applying the technique, two issues must be settled:

- Which functions to memoize on? In [Kho90] some syntactic criteria for deciding when memoization will be useful are given, at the same time exhibiting a method for "compile time garbage collection" of obsolete rules.
- Which kind of equality to use when deciding whether a function has been called with the "same" argument before? The natural choice is "structural equality", but this can lead to very time-consuming comparisons and in the case of lazy data structures even cause infinite loops. Therefore, [Hug85] suggests to use "equality of pointers" instead (of course, then less computation will be shared).

Memoization is a top-down method – its bottom-up counterpart is often termed *tabulation* and is treated e.g. in [Bir80] and [Coh83][1].

[1]In [Bir80] the top-down method is termed *exact tabulation* (as only the rules needed are generated), whereas the bottom-up method may give rise to *overtabulation*. In [Coh83] the top-

Example 2.1.1 Consider the fibonacci function

fib(0) $\rightarrow$*1*

fib(1) $\rightarrow$*1*

fib(n) $\rightarrow$*fib(n-1) + fib(n-2)* *for* $n \geq 2$.

a suitable representation of which will constitute the rules in $\mathcal{R}_0$. Memoization-based evaluation of *fib*(n) amounts to creating a *n-1* level transition system where $\mathcal{R}_1$ consists of the transition from *fib(2)* to 2; where $\mathcal{R}_2$ consists of the transition from *fib(3)* to 3, etc. □

One must realize that memoization (in the form sketched above) is not able to catch all repeated computations – if e.g. *f* first is called with (α,β_1) as argument and then with (α,β_2) as argument $(\beta_1 \neq \beta_2)$, memoization will give us nothing, even though a lot of computation in *f* may depend on its first argument only. This suggests that it might be useful to memoize on "smaller units of computation" – as will be done in section 3.1.

2.1.2 Unfold/fold transformations

The unfold/fold framework for program transformation dates back to [BD77] and has since been the subject of much interest, primarily aimed at making the process of finding "eureka"-definitions more systematic, e.g. [NN90], [PP90], [PP91b], [PP92]. Also supercompilation [Tur86] can be seen as a variant over the concept.

The process of *first* transforming a source program into a target program by the unfold/fold method and *then* running the target program can be considered as implementing a 2-level system[2] – the target program being the rules at level 1 – by the bottom-up approach. No particular requirements exist concerning the form of the level 1-rules.

Unfold/fold transformations are typically done manually, thus eliminating the risk (otherwise potentially present in bottom-up approaches) of generating infinitely many rules (or looping while generating a rule).

down method is termed *the large-table method*, whereas the bottom-up method is termed *the small-table method* (as rules can be discarded when they have been used to generate the desired higher level rules).

[2]This does not model all applications of the unfold/fold technique, since it may happen that a rule derived during transformation is unfolded/folded against later on – thus 3 levels are present. However, it will always be possible to describe an unfold/fold transformation as a *finite sequence* of "2-level" transformations.

On the other hand, in e.g. [Wad90], [PP90] and [PP91b] mechanizable strategies are given which are guaranteed to terminate for certain kinds of source programs.

Example 2.1.2 Consider the function *f* defined by[3]

f([ ]) →[ ]

f(a::x) →b ::f(x)

f(b::x) →c ::f(x)

f(c::x) →a ::f(x)

and suppose this program is often used to evaluate expressions containing subterms of the form *f(f(t))*. Then it might be a good idea to introduce the *eureka definition*

g(x) →f(f(x))

and replace such subterms with *g(t)*.

> *Remark:* this definition is to be considered as an extra level 0-rule, *not* as a level 1-rule. The justification for this is that *g* does not appear in the source program, hence we – without causing ambiguity – can add the definition to the source program *and then* start the transformation process on the *modified* source program.

We have to derive level 1-rules for *g*; first we look at the term *g([ ])*. By one unfolding, this yields *f(f([ ]))*; by one more unfolding, this yields *f([ ])*; and by one more unfolding we end up with [ ] – enabling us to store the level 1-rule

g([ ]) →[ ]

Next, we consider the term *g(a::x)*. By one unfolding, this yields *f(f(a::x))*; by one more unfolding, this yields *f(b::f(x))*; and by one more unfolding we get *c::f(f(x))*. Finally, this can be folded back to *c::g(x)* giving the level 1-rule

g(a::x) →c::g(x)

In an analogous way, we get the level 1-rules

[3]Here *a*,*b* and *c* are constants, *x* is a variable, [ ] is the empty list and :: is the list constructor.

g(b::x) →a::g(x)

g(c::x) →b::g(x)

and the four level 1-rules for *g* constitute the target program. □

Some noteworthy points concerning the unfold/fold technique:

1. In most applications of the technique, the process of folding an expression e into another expression e' is conceptually equivalent to considering e' as an *abbreviation* of e. Referring back to example 2.1.2, if one considers *g(x)* as an abbreviation of *f(f(x))* one can *by unfolding alone* obtain the level 1-rules

   f(f([ ])) →[ ]

   f(f(a::x)) →c::f(f(x))

   f(f(b::x)) →a::f(f(x))

   f(f(c::x)) →b::f(f(x))

   which determine the same "flow of control" as the target program involving *g* – however, if implemented naively is less efficient.

   In the theory developed in chapter 4, only unfolding will be modeled – the discussion above suggests that this is no serious restriction. Moreover, in chapter 8 (when a logic language is treated) folding is handled explicitly.

2. The reason why we can expect the transformation in example 2.1.2 to improve efficiency is that the level 1-rules represent a "short-cut" in the computation process: instead of unfolding *f* twice *g* is unfolded once. This suggests that the speedup will be roughly a factor two. An attempt to formalize this intuition is made in chapter 4.

3. One must be careful not to decrease the termination domain – in example 2.1.2, this would have happened if we after having unfolded *g(a::x)* into *f(f(a::x))* immediately folds back into *g(a::x)* yielding the level 1 rule

   g(a::x) →g(a::x)

Conditions to prevent this from happening are given in [Kot85]. In chapter 4 the problem is addressed anew, with the aim of giving more intuitively understandable conditions than in [Kot85]. For a logic language, the issue is treated in depth in chapter 8.

4. In the unfold/fold framework one usually also – apart from doing unfolding and folding – is allowed to perform various algebraic manipulations. In section 4.8.3 it is discussed how this fits with the multilevel system view.

2.1.3 Partial evaluation

Partial evaluation (PE for short) can be viewed as a special case of the unfold/fold technique, where the (level 1) rules are of form

$f(\alpha, y) \rightarrow \varepsilon(y)$

with α a constant and $\varepsilon(y)$ an expression with y as only free variable. We say that f has been specialized wrt. α, and that the first argument of f is *static* and the second *dynamic* (of course the above can be generalized to an arbitrary number of static/dynamic arguments)[4]. To generate such a rule may be a good idea if some of f's computation can be done even if only the first argument is known, cf. the discussion at the end of section 2.1.1.

Example 2.1.3 Consider the Ackerman function, represented by the level 0 rules

$ack(0,n) \rightarrow n+1$

$ack(m,0) \rightarrow ack(m-1,1)$ for $m > 0$

$ack(m,n) \rightarrow ack(m-1,ack(m,n-1))$ for $m > 0$, $n > 0$

If *ack* is partially evaluated wrt. its first argument being 2, the following level 1 rules (specialized versions of *ack*) will typically be generated:

$ack(2,0) \rightarrow ack(1,1)$

$ack(2,n) \rightarrow ack(1,ack(2,n-1))$ for $n > 0$

$ack(1,0) \rightarrow ack(0,1)$

$ack(1,n) \rightarrow ack(0,ack(1,n-1))$ for $n > 0$

[4]The concept of PE may be extended a bit, as e.g. in [Tak91] where "context" is taken into account, and in [BCD90] where a function can be specialized wrt. an argument satisfying some predicate.

If we introduce *ack2*, *ack1* and *ack0* such that *ack2(n)* is an abbreviation (cf. the discussion in section 2.1.2) of *ack(2,n)* etc, the level 1 rules will take the form

$ack2(0) \rightarrow ack1(1)$

$ack2(n) \rightarrow ack1(ack2(n\text{-}1))$ *for* $n > 0$

$ack1(0) \rightarrow ack0(1)$

$ack1(n) \rightarrow ack0(ack1(n\text{-}1))$ *for* $n > 0$

$ack0(n) \rightarrow n+1$

and will thus constitute a “self-contained” target program.

The only computation which has been done at PE-time is the evaluation of $m - 1$ for $m = 1, 2$ (and the pattern matching wrt. the first argument) – not enough to reduce the enormous complexity of *ack*! □

The concept of partial evaluation dates back to Kleene's *smn*-theorem from recursion theory (where efficiency improvement was no concern); and in the recent decade the field has evolved tremendously. For a comprehensive treatment of central aspects of PE as well as a historical survey, see [JSS89].

In contrast to most (general) unfold/fold systems, PE is usually intended to be done automatically. As the bottom-up approach is used this implies the risk of non-termination, and in fact the great majority of existing partial evaluators may loop (even when doing PE on programs which themselves terminate). Two sources for non-termination exist:

1. when generating the code for a specialized function, an attempt is made to unfold infinitely often.
2. an attempt is made to specialize a function with respect to infinitely many values.

These problems have been attacked in several ways, e.g.:

- In [Ses88] (1) is avoided by testing for cycles in the call graph (this technique being potentially very space consuming), but (2) remains a possibility.
- In SIMILIX [BD91] one decreases the risk of (1) by not unfolding dynamic tests, i.e. tests whose outcome cannot be decided by the

static arguments alone. However, as all specialized versions possibly needed have to be generated in advance – including some which turn out to be not needed – (2) may occur and actually also (1).

- In [Hol91] an analysis for ensuring that (2) cannot happen is given. If the analysis reveals that (2) *might* happen, one will have to make some static arguments dynamic (thus giving rise to less sharing of computations).

- In [Sah91] a partial evaluator for full PROLOG is given which does not violate termination properties. A number of ways to ensure this are proposed, one of which in the functional context translates into putting an upper limit on the number of specialized versions of each function.

In section 5.2 we shall present a *top-down* version of PE and prove it to preserve termination properties (i.e. terminate unless the source program itself loops).

Chapter 3

Various Degrees of Sharing

When evaluating an expression in a functional language (e.g. the λ-calculus [Bar84]), some subcomputations may have to be done several times, this being a cause of inefficiency. Consequently, several techniques have been devised to increase the amount of sharing. Below, we are going to list some of these techniques in order of increasing sophistication.

First some terminology (from the λ-calculus): a redex met when following the *leftmost* path from a node is called a *spine redex* (of this node). For instance, in $(\lambda x.((\lambda y.y)e_1)e_2)e_3$ the expression itself is a spine redex (the topmost one) and $(\lambda y.y)e_1$ is a spine redex (the bottommost one). When no spine redices exist, the expression is said to be in head normal form. Notice that it is always safe to reduce a spine redex of e, in the sense that if e reduces to a head normal form then this reduction involves the reduction of all spine redices of e [BKKS87, theorem 4.9]. A reduction which reduces the topmost spine redex is termed a *normal order reduction.*

Lazy evaluation

One can work with “DAG”s instead of trees, i.e. allow a subexpression to be shared among several expressions. When the normal order strategy is applied, this amounts to using lazy evaluation instead of simple CBN.

Fully lazy evaluation

The use of DAGs is not sufficient to avoid duplication of computations. To see this, consider the λ-expression E defined by

$E = (\lambda f. + (f\ 2)(f\ 3))(\lambda y. + (h\ 4)y)$

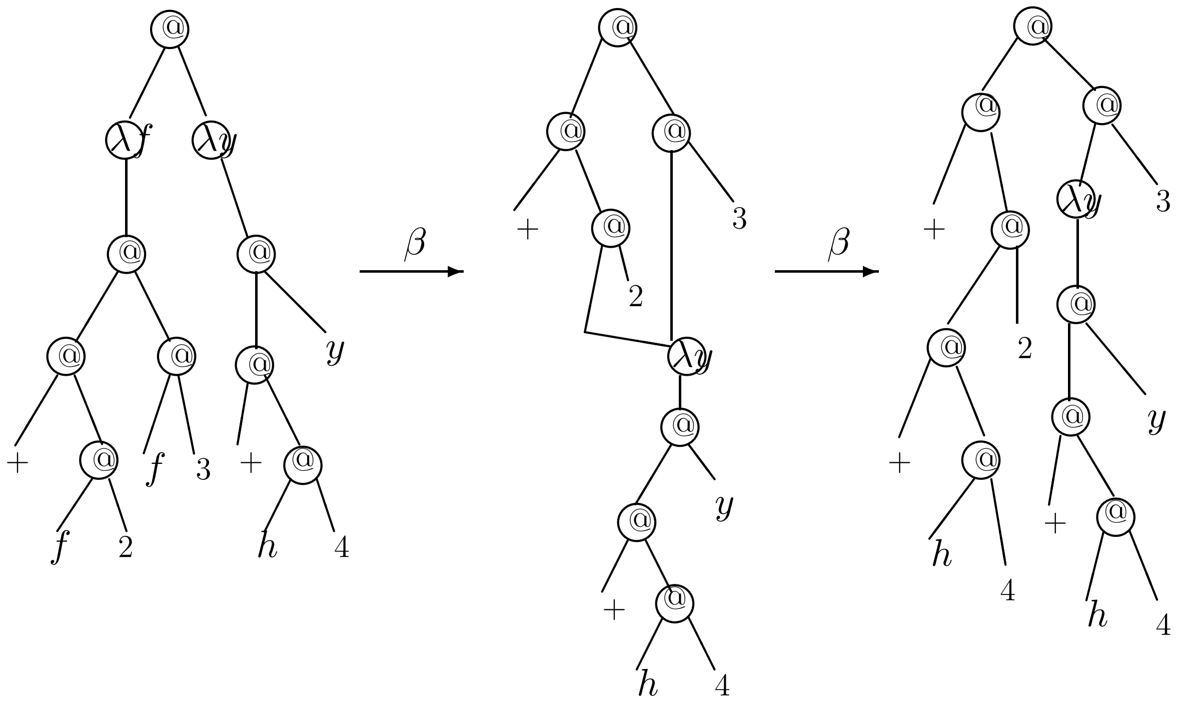


Figure 3.1: Naive β-reduction of E

where h is some (expensive) built-in function.

A *naive* evaluation of E will give rise to a sequence of β-reductions the first two steps of which are depicted in figure 3.1.

When doing the first β-reduction, the subexpression $\lambda y. + (h\ 4)y$ can be shared since we are working with DAGs. But when performing the second β-reduction, i.e. when applying the abovementioned expression to 2, everything below the "λy" (i.e. the subexpression $+(h\ 4)y$) is *duplicated* (since 2 has to be inserted on y's place in one of the copies but not in the other). Hence h 4 will be evaluated twice.

However, one can do better: actually there is no need to copy $+(h\ 4)$, since $+(h\ 4)$ does not contain y free – in [Jon87, p. 246] this observation is attributed to [Wad71]. An implementation clever enough to avoid such unnecessary copying is termed *fully lazy* [Jon87, p. 210].

Usually, one does not implement functional languages by means of β-reductions of λ-expressions – instead, one transforms into *supercombinators* by *lambda-lifting* ([Jon87]). By doing so, E might be *naively* translated into the supercombinator program

$E = L\ M$

$L\ x = +\ (x\ 2)(x\ 3)$

$M\ y = +\ (h\ 4)\ y$

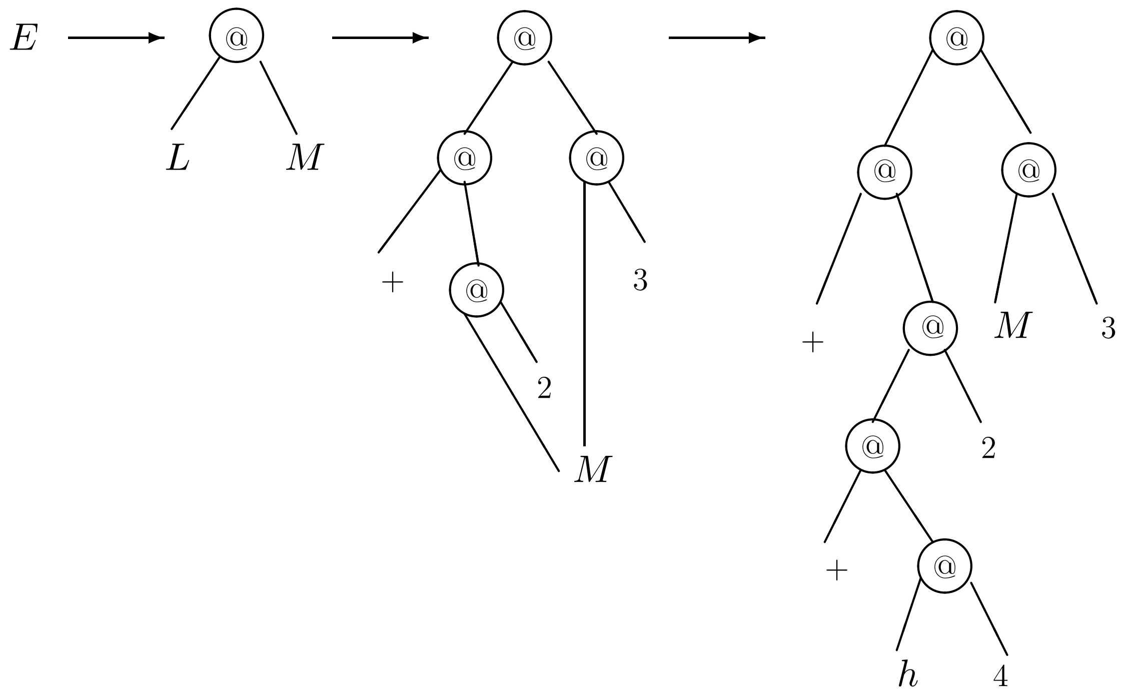


Figure 3.2: Combinator based reduction of E

This program will be evaluated as depicted in figure 3.2. As the expression h 4 is "hidden" in the definition of the supercombinator M, h 4 will be evaluated each time M is applied – i.e. twice.

Again, one can do better: one can modify the lambda-lifting algorithm so it abstracts away the *maximal free expressions*, as done in [Hug82]. Then the resulting supercombinator program will be fully lazy in the sense that it has the "same" sharing properties as the original λ-expression when evaluated by a fully lazy implementation. Now E is translated into (as the maximal free expression h 4 is abstracted away from the definition of M)

$E = L\ (M\ (h\ 4))$

$L\ x = +\ (x\ 2)(x\ 3)$

$M\ x\ y = +\ x\ y$

E can now be evaluated as depicted in figure 3.3: only one copy of h 4 then ever exists.

In some sense, fully lazy evaluation guarantees that redices *present in the source program* are not copied [HG91]. However, when looking at a program written in a high-level functional language, it is a rather tricky issue to see which subcomputations are shared (when the program

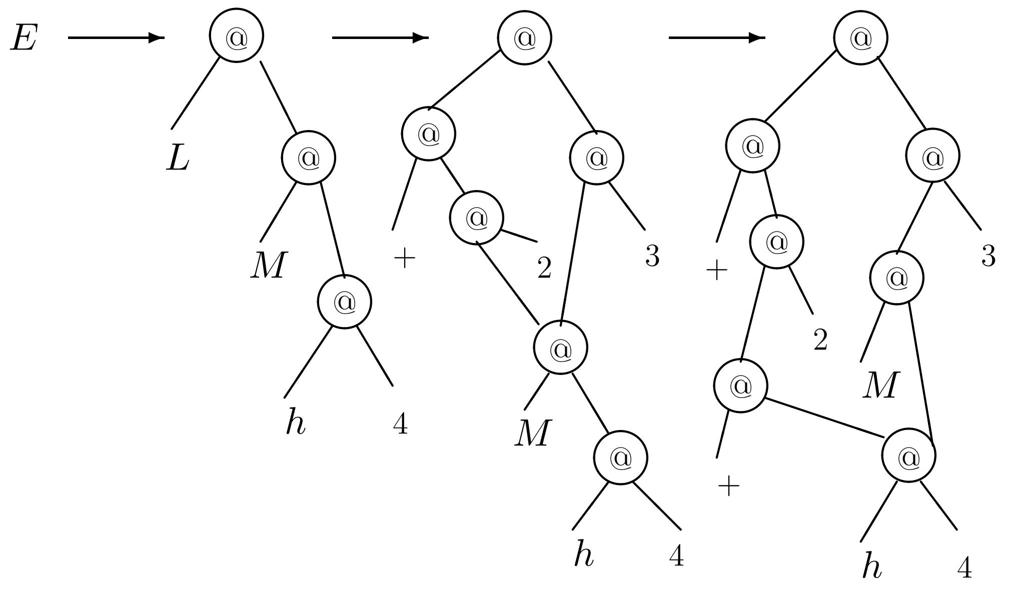


Figure 3.3: (Fully lazy) combinator reduction of E

is translated into the λ-calculus). In fact, the discussion in [Jon87, chap. 23] ends up with (p. 400)

> We conclude that it is by no means obvious how lazy a function is, and that we do not at present have any tools for reasoning about this. Laziness is a delicate property of a function, and seemingly innocuous program transformations may lose laziness.

Moreover, as pointed out in [FS91] we have

Fully lazy evaluation is not *fully* lazy

and may in fact be exponentially bad. To see this, consider – as in [FS91] – the family of λ-expressions given recursively by

$$\begin{aligned} A_0 &= \lambda x.I \\ A_n &= \lambda h.(\lambda w.wh(ww))A_{n-1} \text{ for } n \geq 1 \\ B_n &= A_n I \text{ for all } n \end{aligned}$$

where $I = \lambda x.x$.

Let us now perform a fully lazy sequence (i.e. the topmost spine redex is reduced, and expressions not containing the bound variable free are not copied) of β-reductions on B_n. We want to show by induction that $A_n x$ reduces to I for any λ-expression x, and at the same time we want to calculate the number of reduction steps $T(n)$. For $n = 0$ the claim is obvious, and $T(0) = 1$. For $n \geq 1$ a key observation is that when A_n is applied to x, everything in the body of A_n is copied except the subexpression A_{n-1} (and except $(w\ w)$) – even by a fully lazy evaluation. This justifies that we can describe the reduction sequence in "linear form":

$$\begin{array}{lll} A_n x & \rightarrow & (\lambda w.wx(ww))A_{n-1} \\ & \rightarrow & A_{n-1}x(A_{n-1}A_{n-1}) \\ & \rightarrow^{T(n-1)} & I(A_{n-1}A_{n-1}) \\ & \rightarrow & A_{n-1}A_{n-1} \\ & \rightarrow^{T(n-1)} & I \end{array}$$

From the above we see that $T(n) = 2T(n-1) + 3$, giving us

$$T(n) = 4 \cdot 2^n - 3$$

This is exponentially bad – to see this, notice that A_n actually β-reduces to A_0 in $4n$ steps: by induction we have

$$\begin{array}{lll} A_n & = & \lambda h.(\lambda w.wh(ww))A_{n-1} \\ & \rightarrow & \lambda h.A_{n-1}h(A_{n-1}A_{n-1}) \\ & \rightarrow^{4(n-1)} & \lambda h.(A_0h(A_0A_0)) \\ & \rightarrow & \lambda h.I(A_0A_0) \rightarrow \lambda h.A_0A_0 \rightarrow \lambda h.I \\ & = & A_0 \end{array}$$

where we exploit that the occurrences of A_{n-1} in $\lambda h.A_{n-1}h(A_{n-1}A_{n-1})$ can be shared. Notice that in the above reduction sequence, only spine redices are reduced.

By reducing something which is not the topmost spine redex (but nevertheless a spine redex), we thus gain an exponential speed-up.

Now let us translate the above example to supercombinators (still following [FS91]): we arrive at

$L\ h\ w = w\ h\ (w\ w)$

$M_n\ h = L\ h\ M_{n-1}$

$M_0\ h = I$

hence B_n is translated to M_nI. Let us show that M_nx for any x reduces to I: for $n = 0$ this reduces to I in 1 step, and for $n \geq 1$ we by induction have

$$\begin{array}{rll} M_n x & \rightarrow & L x M_{n-1} \\ & \rightarrow & M_{n-1} x (M_{n-1} M_{n-1}) \\ & \rightarrow^* & I(M_{n-1} M_{n-1}) \\ & \rightarrow & (M_{n-1} M_{n-1}) \\ & \rightarrow^* & I \end{array}$$

As was to be expected, this is isomorphic to the fully lazy β-reduction of $A_n x$ – still run time is exponential.

In the β-reduction world, we overcome the exponential complexity by reducing a spine redex which is not the topmost, i.e. to reduce even though not all variables are known. But this is just the essence of partial evaluation! Accordingly, in the supercombinator world the analogous path to success (being inspired by the concept of multilevel systems introduced in chapter 2) is to combine partial evaluation and memoization into a reduction strategy to be termed *ultimate sharing*, to be sketched and discussed in the following section. An abstract machine for ultimate sharing is defined in chapter 5.

3.1 Ultimate Sharing

Let us sketch how one by means of ultimate sharing may evaluate B_n – for simplicity, let us assume $n = 3$.

1. Starting with the expression $M_3 I$, we apply the equation for M_3 to arrive at $L I M_2$. Applying the equation for L, we get $M_2 I (M_2 M_2)$.
2. Perhaps inspired by the fact that M_2 in the expression above is applied to two (distinct) arguments, we decide to *partially evaluate* M_2 (wrt. to zero known arguments!). Thus, instead of looking at the expression $M_2 I$ we look at the more general expression $M_2 v$, where v is a "variable". The aim of this is to "factor out" the argument-independent computation of M_2, so this computation can be shared among various invocations of M_2.
   (a) The expression $M_2 v$ is via $L v M_1$ reduced to $M_1 v (M_1 M_1)$. In this expression, we continue working on $M_1 v$.
      i. $M_1 v$ reduces via $L v M_0$ to $M_0 v (M_0 M_0)$. The applications of M_0 can be carried out directly, giving us I via $I(M_0 M_0)$ and $M_0 M_0$.

  ii. We have thus found out that $M_1 v$ reduces to I – in 5 steps. This fact is now stored as a rule at level 1 (since no rules, except for the "predefined" at level 0, have been exploited to deduce it).

 (b) We return to the evaluation of $M_2 v$, which we so far have reduced to $M_1 v(M_1 M_1)$. Due to the rule just derived, we infer that this reduces to $I(M_1 M_1)$. This in turn reduces to $M_1 M_1$, and now we can gain the benefits of *memoization*: by retrieving the level 1 rule just stored we see that this reduces to I.

 (c) We have thus found out that $M_2 v$ reduces to I – in 5 steps, excluded the steps needed for deriving the rule for M_1. Again, we store this fact as a rule; this time at level 2, since a level 1 rule was used for the deduction.

3. We return to the evaluation of $M_3 I$, which we so far have reduced to $M_2 I(M_2 M_2)$. Making use of the rule for M_2 twice, this reduces - via $I(M_2 M_2)$ and $M_2 M_2$ – to I.

4. Thus, we have finally solved the problem: $M_3 I$ has been reduced (at level 3) to I – in 5 steps, excluded the steps needed for deriving the rule for M_2.

It should be emphasized that "ultimate sharing" is a *non-deterministic* reduction strategy. In order to make it deterministic (for implementation purposes), one has to settle on the following issues:

- when to start looking at a more general expression.
- which more general expression to look at (in general, there will be several possibilities).
- when to stop the current reduction sequence and store the result obtained so far.
- when to examine whether previously stored rules are useful.

Conceptually one can imagine these decisions to be made by an oracle; in reality these decisions will be made *either* by the user *or* by means of some analysis/heuristics, *either* beforehand *or* during evaluation. See section 9.1 for a brief discussion on how to perform such an analysis.

3.2 Is this really ultimate?

Just as "fully lazy" evaluation is not *fully* lazy, one may argue that "ultimate sharing" is not *ultimate* sharing. In particular, a machine able to perform *induction proofs* will do better in the example above: by proving the theorem $\forall n, x : M_n x \rightarrow^* I$ it is possible to evaluate B_n using a *constant* number of steps. So our usage of the term "ultimate sharing" is perhaps somewhat misleading...

3.3 Two kinds of sharing computations

Notice that we have introduced *two* ways of reusing computations: *sharing* (of nodes in a graph) and *memoization* (checking if a similar subgraph has been met before). The reader may wonder if one should not apply Occam's razor and abandon one of these features. However, the intuition is that

- as sharing of nodes comes for free in an efficient implementation, it is performed implicitly;
- as memoization is potentially expensive it is performed explicitly, on selected expressions only.

3.4 Applicability of ultimate sharing

The example given shows that an evaluation strategy based on ultimate sharing may be exponentially better than one just being fully lazy. This example may seem contrived, but in chapter 6 we shall see a "realistic" program which runs exponentially faster if executed by an ultimate sharing strategy.

Ultimate sharing can be considered as a *top-down* implementation of a multilevel transition system, cf. the discussion p. 20. In particular, a top-down version of partial evaluation can be obtained, as will be described in section 5.2. Just as (bottom-up) PE often is used for compilation purposes (by doing PE on interpreters) [JSS89], in [AT89] the idea above was used for implementing "lazy and incremental" compilation (by evaluating an interpreter using a strategy based on ultimate sharing).

On the other hand, it should also be clear that for most applications ultimate sharing will be a far too heavy tool (unless in a very restricted form), the implementation overhead being prohibitively expensive.

3.5 Ultimate sharing and related concepts in the literature

The idea of improving the efficiency of an algorithm by reusing previously computed results is, of course, by no means new – this is just the philosophy behind *dynamic programming* (see e.g. [Har87]). However, when the algorithm to be improved is one for implementing a functional language we so to speak step up one level, since to make this particular algorithm faster (for some inputs) amounts to making *many* programs faster.

Concepts resembling ultimate sharing seem to have come up rather independently – in various disguises – several places in the recent years:

- in [AT89] it is denoted *partial memoization* and is worked out for an eager, first-order combinator language – a system was implemented (guided by instructions from the user) and proved correct.
- in [HG91] the concept (which is there christened *complete laziness*) is elaborated in a higher-order setting; one of the main purposes is to argue that partial evaluation for higher order languages is able to achieve strictly more sharing than "full laziness" – their treatment being a source of inspiration for this chapter. Further, it is sketched how to implement complete laziness via the underlying evaluation mechanism in a lazy language (thus, in effect *all* subexpressions are memoized upon).
- in [Gru] a reduction strategy for the pure λ-calculus, termed *call-by-mix*, is proposed: the redex selected for reduction is the bottommost spine redex; and when doing this reduction some other tricks are employed in order to keep subexpressions shared as long as possible. The properties of this strategy would surely be worth a closer study; Grue (personal communication) has not investigated the issue further.
- in [Lam90] (which [GAL92] attempts to elaborate on) an algorithm for graph reduction of λ-expressions is given which attempts to

avoid duplication of computation by keeping track of how redices propagate – on the other hand, "accidental sharing" (that is redices which are identical but not copies of the same original redex) is not detected. The trick employed is to let graphs contain not only "fan-ins" but also "fan-outs", i.e. a node may point at several other nodes, the one to actually "choose" given from the context. Thus graphs which differ only "far below" can share representation.

- in [Kah92] a system implementing not only a "hashing cons" but also a "hashing apply" is described, i.e. the mapping from nodes in the heap into terms is guaranteed to be injective – thus also "accidental sharing" is detected, and our distinction (section 3.3) between sharing and memoization collapses. On the other hand, due to the absence of partial evaluation a hashing apply is not as powerful as ultimate sharing: if we for instance had definitions $M\ x\ =\ L\ x$; $L\ x\ =\ 7$ then one by the latter method is able to recognize that $M\ x\ = 7$ for all x, a fact which cannot be detected by the former method.

3.6 Choice of framework

What we have seen so far suggests that efforts taken to maximize sharing can be done within several models of computation, and that there is a close correspondence between reduction strategies in the various frameworks[1]. As usual when different frameworks are involved, it is rather difficult to compare the effects formally, and we shall not attempt to do so. In the subsequent chapters, we settle upon one framework. For some pragmatic reasons, I have chosen to use a language with named combinators and pattern matching, implemented by means of graph reductions, instead of the λ-calculus:

- One does not have to worry about free variables getting bound by β-reduction (this problem being a main reason why the "bottom-most spine redex" strategy may be considered unfeasible in practice [Jon87, p. 200]).

[1]For SKI-combinators, as defined in [Tur79], the trick to obtain the effects of fully lazy evaluation is [Jon87, p. 267] to incorporate the rule $S\ (K\ p)\ (K\ q) \rightarrow K\ (p\ q)$.

- If one wants to include constants (like **if** and **plus**), what we surely will do for modeling practical applications, the language will anyway get the flavor of a combinator language (there will be δ-rules etc).

Chapter 4

A model for a functional language or **how to get more than a constant speedup**

This chapter will be devoted to formalizing the concept of multilevel transition system, as introduced in chapter 2, in a functional setting. As this can be considered the main chapter of the thesis, a larger overview is appropriate:

- Section 4.1 sets up the basic machinery of graphs and graph reductions. It should be clear from section 3.1 that it will be most useful to be able to express that a graph G_1 is more general than a graph G_2, likewise we want to express that a graph G_1 rewrites to a graph G_2. The latter will be modeled by the existence of a *reduction* (cf. definition 4.1.9) from G_1 to G_2; the former will be modeled by the existence of a *specialization* (cf. definition 4.1.10) from *the sum of* (cf. section 4.1.2) G_1 and some graph G to G_2.

  Also, it should be clear from chapter 2 and section 3.1 that it will be useful to make sure that "if there is a reduction r from graph g to g', and g is more general than G, then r gives rise to a reduction r' from G to some new graph G', where g' is more general than G'". This will be modeled by a *pushout* (cf. definition 4.1.19), this being the standard way of expressing such "reduction within a context" (see e.g. [ENRR87]).

  Section 4.1 is rather long and technical, especially the proof that the pushout actually exists (section 4.1.5). However, in order to read the rest of this chapter it will be sufficient if one understands how graphs, specializations, reductions, sums and pushouts are defined

– furthermore, one should be aware of the algebraic laws stated in section 4.1.4.

We do not in any way claim our approach to be original, in particular the approach in [Rao84] is rather close to ours. However, the development (especially in section 4.3) is given a new particular twist in order to suit our (later) purposes.

- The graphs in section 4.1 have all been *singlelabeled*; in section 4.2 *multilabeled* graphs are introduced – enabling one to express e.g. that the value of one node is twice the value of another; or that a given node can assume any value except 7. It is sketched how the development from section 4.1 carries over; for instance pushouts will still exist – this material may be skipped by the reader.

- In section 4.3 we equip graphs with *demand functions*, which for each node in the graph estimate "how far" it (at least) must be reduced – 2 means that the node must be reduced to "normal form", 1 means that the node (at least) must be reduced to "weak head normal form", and 0 means that nothing can be said for sure. Of course, some well-formedness criterion must be met (definition 4.3.2).

  The purpose of introducing a demand function is to capture the property of a "redex the reduction of which is needed" in a *syntactic* way, enabling an efficient implementation of an "optimal evaluation strategy".

  It is sketched how the development from section 4.1 carries over; for instance pushouts will still exist – this material is rather long and may be skipped by the reader. Finally the notion of a *result node* can be defined (section 4.3.1) – such a node has, of course, demand 2.

- Section 4.4 considers the level 1 transitions and investigates their properties. Of special interest is *normal order reductions*, reductions where only redices with demand ≥ 1 are reduced – graphs with no such redices are said to be in *normal form* (section 4.4.1). In order for interesting properties to hold, it is necessary to put some rather tight requirements on the form of the level 0 rules.

  Theorem 4.4.4 now states that the transition system is confluent, and theorem 4.4.14 states that normal order reduction is optimal in the sense that if some reduction reaches a normal form in c steps any normal order reduction reaches an "equivalent" normal form in $\leq c$ steps.

- Section 4.5 completes the definition of the multilevel transition system, by considering the level i transitions. It is shown that they satisfy some "closedness" properties.
- Section 4.6 states results concerning "correctness", i.e. whether one gets the same result by working at level i as one does when working at level 1, and at the same time addresses the question how much speedup one can expect to gain (at most) by working at level i instead of level 1. In particular,
  - theorem 4.6.3 expresses *partial correctness* and at the same time gives a *speedup bound*: there exists a constant k (dependent on the rules) such that if reduction at level i reaches a normal form in c_i steps then (normal order) reduction at level 1 reaches an equivalent normal form in $\leq kc_i$ steps;
  - theorem 4.6.4 can be interpreted as saying that by having an upper bound on the number of levels employed in a multilevel system, one at most gains a *polynomial* speedup;
  - theorem 4.6.7 gives criteria for *total correctness*, i.e. that if reduction at level i loops then also (normal order) reduction at level 1 loops – the trick is to ensure that each rule represents some "progress".
- In section 4.7 examples are given showing how the results from section 4.6 apply to "multilevel systems as one encounters them in the real world", cf. section 2.1 – that is, we examine memoization, unfold/fold transformations and partial evaluation. Section 4.7.4 is devoted to a discussion of the relevance of our complexity measure. Finally, section 4.7.5 briefly *attempts* to justify that we later on mostly will represent graphs as terms.
- Section 4.8 is (in my opinion!) perhaps the most interesting part of the chapter. Here we factor our some reasons why a program optimization technique may yield *more than* a constant speedup (or more than a polynomial speedup), namely

  - that the strategy used at level 1 is not optimal (if e.g. "call-by-value" is used) and transformation "simulates" an optimal strategy (section 4.8.1);
  - that some sharing is introduced during the transformation process (section 4.8.2);
  - that some "laws" (or lemmas) are proved and exploited during transformation (section 4.8.3).

  Finally (section 4.8.4), we will explain the remarkable speedup achieved in [PB82] (the fibonacci function transformed from being exponential into being logarithmic) in terms of the above "factorization".

- Section 4.9 attempts to put the present work into perspective by describing (in more or less detail) other approaches from the literature.

- Finally, in section 4.10 some ways in which our model could be extended are discussed – this includes allowing reductions to take place in *parallel*; to model *call-by-value* reduction; and to model *garbage collection*.

4.1 Graphs and graph reductions

In the multilevel transition systems to be treated, the configurations will be graphs – motivated by the fact that functional languages mostly (as indicated in chapter 3) are implemented by means of graph reduction [Jon87].

We will assume a set of symbols S, each $s \in \mathsf{S}$ having a fixed arity $Ar(s)$. The intuition is that if a node is labeled s then this node has $Ar(s)$ children. Also, to each $s \in \mathsf{S}$ we assign a "function arity", written $Far(s)$. S might include the natural numbers with arity 0 (and function arity 0), the list constructor :: with arity 2 and (arbitrarily) function arity 0, the plus operator + with arity 0 and function arity 2 etc, together with "user defined function symbols" with arity 0.

> *Remark:* We do not impose any "type discipline" in our model, as our concern is the operational behavior. Then,

since "badly typed programs may go wrong" (paraphrasing [Mil78]), "stuck configurations" may arise.

Definition 4.1.1 A *graph* is a triple $(N, \mathcal{L}, \mathcal{S})$ where

- N is a (finite) set of *nodes*, N being the union of three disjoint sets:
  - The set of *active nodes*, denoted A – to be thought of as application nodes. A node named a (possibly with some subscript) is implicitly assumed to be active.
  - The set of *passive nodes*, denoted P – to be thought of as constructor nodes. A node named p (possibly with some subscript) is implicitly assumed to be passive.
  - The set of *virtual nodes*, denoted V – to be thought of as "holes" where other graphs are to be plugged in. A node named v (possibly with some subscript) is implicitly assumed to be virtual.
- $\mathcal{L}$ is a *labeling function* from P to S. We will often write $Ar(p)$ instead of $Ar(\mathcal{L}(p))$. (When drawing pictures, active nodes will be labeled by a special symbol @ .)
- $\mathcal{S}$ (the *successor function*) is a mapping which to each $p \in P$ assigns a mapping from $\{1 \ldots Ar(p)\}$ to N, and which to each $a \in A$ assigns a mapping from $\{1, 2\}$ to N. We will often write $\mathcal{S}(n, i)$ for $\mathcal{S}(n)(i)$ or simply write "the *i'th* child of n". By assigning each active node arity 2, we can consider $\mathcal{S}$ as a mapping which to each $n \in N \setminus V$ assigns a mapping from $\{1 \ldots Ar(n)\}$ to N.

□

Definition 4.1.2 We define a relation $\ll$ by stipulating $n_1 \ll n_2$ iff $n_1 \in rg(\mathcal{S}(n_2))$. $\prec$ is the transitive closure of $\ll$, $\preceq$ is the reflexive closure of $\prec$. □

Given active node a, we say that $n = Sp(a, i)$ (n is i steps down the spine of a) if either $i = 1$, $n = \mathcal{S}(a, 1)$ or there exists active node a' with $a' = \mathcal{S}(a, 1)$, $n = Sp(a', i - 1)$.

Definition 4.1.3 An active node a is said to be a *genuine partial application* if there exists i and passive node p such that $p = Sp(a, i)$, and such that $Far(\mathcal{L}(p)) > i$. □

Definition 4.1.4 An active node a is said to be a *partial application* if it is a genuine partial application or there exists i and virtual node v such that $v = Sp(a, i)$ $\square$

So if a is a partial application, $\mathcal{S}(a, 1)$ is either a partial application, virtual or passive. And if a is a genuine partial application, $\mathcal{S}(a, 1)$ is either a genuine partial application or passive.

Example 4.1.5 As $+$ has function arity 2, an active node a is a genuine partial application if $\mathcal{S}(a, 1)$ is labeled $+$. On the other hand, if $a = \mathcal{S}(a', 1)$ for some a' then this a' is *not* a partial application, since now *all* arguments to $+$ will be present. $\square$

The intuition behind giving partial applications an explicit status in the theory is that these nodes are guaranteed never to be "redices" - however, non-genuine partial applications may become redices "if placed in some context".

4.1.1 Morphisms between graphs

A *morphism* from G_1 to G_2 is a (total) mapping from the nodes of G_1 to the nodes of G_2. If m_1 is a morphism from G_1 to G_2 and m_2 is a morphism from G_2 to G_3, one by functional composition can define $m_1 \star m_2$, a morphism from G_1 to G_3. Likewise, one for any G can define the morphism id_G as the identity mapping.

> *Remark:* If one had defined morphisms to be *partial mappings*, *garbage collection* would be modeled. The theory is being somewhat complicated by such an approach – in section 4.10 we briefly sketch how to pursue this line.
>
> When illustrating morphisms by means of figures, we will often garbage collect some of the nodes (so one can concentrate on the essential parts of the morphism). It should be rather easy to "complete" such figures.

We will be interested in several kinds of morphisms, e.g. *specializations* and *reductions*, to be motivated briefly:

- A specialization from G_1 to G_2 models that G_1 is "more general" than G_2. For instance, the graph (represented by the term) *f(v)* (v being a virtual node) is more general than the term *g(f(3),3)*, as

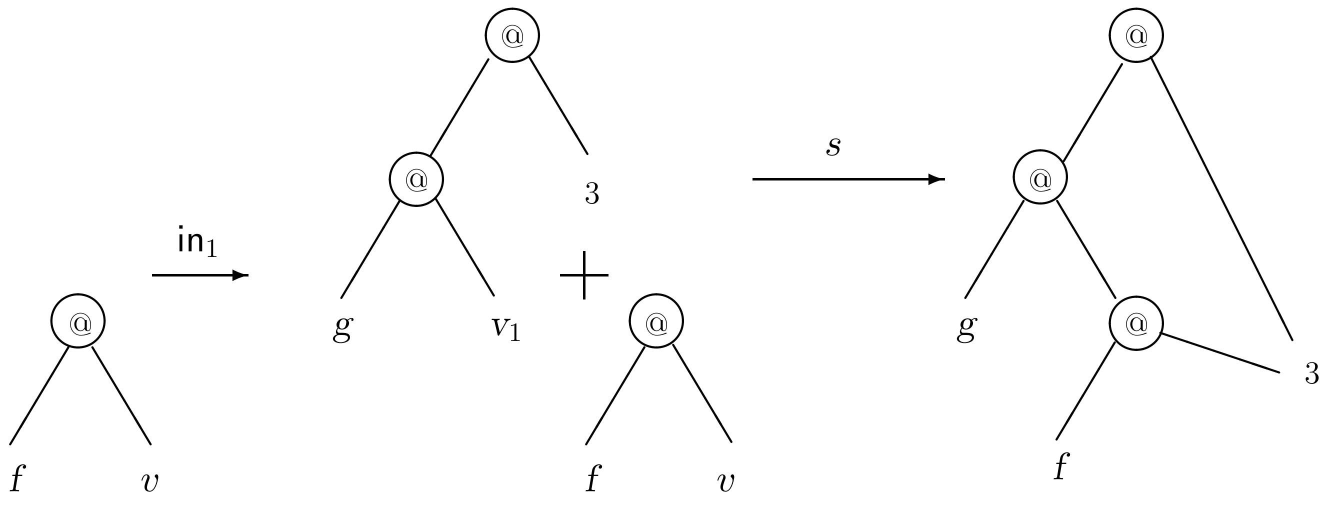


Figure 4.1: A specialization.

1) *v* has been instantiated to 3 in the latter graph 2) in the latter graph, some context (*g*) is included. Accordingly, as depicted in figure 4.1, there is a specialization *s* from G_1 to *g(f(3),3)* where G_1 is the "sum" of *f(v)* and the graph *g(*v_1*,3)*. *s* maps e.g. *v* into 3 and v_1 into the bottommost application node.

- A reduction from G_1 to G_2 models that G_1 "is reduced to" G_2 during the process of evaluating G_1. For instance, one could imagine that *f(x)* $\rightarrow$*x* + *x* is a clause in the source program. Then there will be a reduction r_1 from *f(x)* to *x* + *x*, as depicted in the left part of figure 4.2. Here r_1 maps the application node in the former graph into the topmost application node in the latter graph. On the other hand, if *f(x)* $\rightarrow$*x* is a clause in the source program then the situation is as depicted in the right part of figure 4.2, where there is a reduction r_2 from *f(x)* to *x*. Here r_2 maps the application node in the former graph as well as the virtual node *v* into the virtual node in the latter graph. In both cases, the *f*-node has been implicitly garbage collected.

Some notation: we say that a morphism m from G_1 to G_2 *respects* a node n iff the following holds[1]: $m(n)$ is the same kind of node (active/passive/virtual) as n; if n is passive then $\mathcal{L}_2(m(n)) = \mathcal{L}_1(n)$ (hence also $Ar_2(m(n)) = Ar_1(n)$); and if n is active or passive then for all

[1][Rao84] uses the terminology that *m is a morphism at n* iff the last two conditions hold, i.e. if labels and successors are preserved.

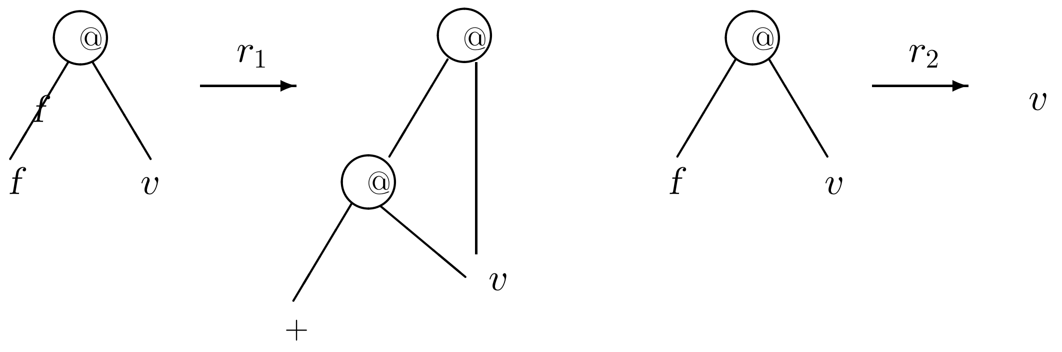


Figure 4.2: Two reductions.

$i \in \{1 \ldots Ar(n)\}$: $m(\mathcal{S}_1(n,i)) = \mathcal{S}_2(m(n),i)$.

Definition 4.1.6 Let m be a morphism from G to G'. We say that m is an isomorphism iff m is bijective and respects all nodes. □

It seems fair to say that if there is an isomorphism from G to G', then "G and G' cannot be distinguished". We say that G and G' are isomorphic.

Definition 4.1.7 Let m be a morphism from G to G'. We say that m is a homomorphism iff m respects all active and passive nodes. □

Lemma 4.1.8 Suppose h_1 is a homomorphism from G_1 to G_2 and h_2 is a homomorphism from G_2 to G_1, such that $h_1 \star h_2 = \mathrm{id}_{G_1}$, $h_2 \star h_1 = \mathrm{id}_{G_2}$. Then h_1 and h_2 are isomorphisms. □

Definition 4.1.9 A reduction r from G to G' is a morphism such that

1. r respects all passive and virtual nodes, and all partial applications.

2. r is injective on virtual nodes, i.e. $r(v_1) = r(v_2)$ implies $v_1 = v_2$.

□

The motivation for 1 is that one, during graph reduction, only performs "non-trivial rewriting" on nodes which are "redices". The motivation for 2 is that one cannot allow two distinct "holes" to be unified - if e.g. the first hole is a placeholder for a passive node and the second hole is a placeholder for a partial application, then a conflict will arise.

Definition 4.1.10 A specialization s from G to G' is a morphism such that

1. s respects all active and passive nodes.

2. If $s(a_1) = s(a_2)$, with $a_1 \neq a_2$, then a_1 and a_2 are partial applications.
3. For all a' in G' there exists an a such that $s(a) = a'$; for all p' in G' there exists a p such that $s(p) = p'$; for all v' in G' there exists a v such that $s(v) = v'$ (this is a stronger requirement than surjectivity).

□

Condition 1 is quite natural; and condition 3 is motivated by our choice to model "G is more general than G'" by the existence of a specialization from *the sum of some* G_1 *and* G to G', rather than by the existence of a specialization from G itself to G' (cf. lemma 4.1.18). Concerning condition 2, the technical motivation is that if $s(a_1) = s(a_2)$, $a_1 \neq a_2$ and a_1 *not* a partial application, then – as a reduction r does not necessarily preserve active nodes which are not partial applications – $r(a_1)$ is "out of control" and may differ "significantly" from $r(a_2)$, thus making it impossible to define the pushout[2]. Pragmatically, the motivation is that it will never pay off to "split" an active node – e.g. when looking at a graph $f(gx)(gx)$ where the two occurrences of gx are shared, it will do no good to look at a more general graph where the two occurrences are not shared.

Observation 4.1.11 It is easily seen that if a is a (genuine) partial application and r a reduction, then $r(a)$ is also a (genuine) partial application.

If $s(a)$ is a partial application then so is a – but a is not necessarily a genuine partial application, even if $s(a)$ is.

On the other hand, if a is a genuine partial application so is $s(a)$ – but $s(a)$ is not necessarily a partial application, if a is only a (non-genuine) partial application. □

Fact 4.1.12 For all G, id_G is a morphism belonging to all kinds (i.e. a reduction, a homomorphism etc). An isomorphism belongs to all kinds of morphisms. All kinds of morphisms are stable under $\star$. □

Proof: The claims concerning id_G are trivial; so is the fact that isomorphisms, homomorphisms and reductions are closed under $\star$. Now let us see that specializations are closed under $\star$:

[2]Cf. the remarks [Rao84, p. 11], to motivate why restrictions somewhat similar to condition 2 have been given.

1 and 3 are trivial. For 2, assume that $s_2(s_1(a)) = s_2(s_1(a'))$ with $a \neq a'$. Since s_1 satisfies 1, $s_1(a)$ and $s_1(a')$ are active. If $s_1(a) = s_1(a')$, we from s_1 satisfying 2 can conclude that a and a' are partial applications. If $s_1(a) \neq s_1(a')$, we from s_2 satisfying 2 can conclude that $s_1(a)$ as well as $s_1(a')$ are partial applications, and hence (by observation 4.1.11) also a and a' are partial applications. □

4.1.2 The + operator

Given two graphs G_1 and G_2, we can define $G_1{+}G_2$ in the standard way[3], i.e. as the "disjoint union" of G_1 and G_2. Let in_1 (a morphism from G_1 to $G_1{+}G_2$) and in_2 be the "injection functions".

Observation 4.1.13 Given morphisms m_1 from G_1 to G_1' and m_2 from G_2 to G_2', there exists a unique morphism m from $G_1{+}G_2$ to $G_1'{+}G_2'$ such that

$$\mathsf{in}_1{\star}m = m_1{\star}\mathsf{in}_1{}', \mathsf{in}_2{\star}m = m_2{\star}\mathsf{in}_2{}'$$

If m_1 and m_2 both are isomorphisms (homomorphisms, reductions, specialization) then also m is an isomorphism (homomorphism, reduction, specialization). □

This m is denoted $m_1{+}m_2$.

As we do not in any way distinguish between isomorphic graphs, we have to check that + "respects" isomorphism. But it is immediate from what is said so far that if G_1 and G_1' are isomorphic, and G_2 and G_2' are isomorphic, then also $G_1{+}G_2$ and $G_1'{+}G_2'$ are isomorphic.

Fact 4.1.14 + is commutative and associative, up to isomorphism. □

Fact 4.1.15 + is a functor, i.e. (with symbols having appropriate functionality):

1. $\mathrm{id}_{G_1}{+}\mathrm{id}_{G_2} = \mathrm{id}_{G_1+G_2}$

[3]Of course, it would be nice to have a categorical definition. However, with the extensions to the model to be presented later on it does not seem quite easy to come up with such a definition – and actually fact 4.1.16 provides the categorical property we shall need.

2. $(m_1{\star}m'_1){+}(m_2{\star}m'_2) = (m_1{+}m_2){\star}(m'_1{+}m'_2)$

$\square$

Proof: For 1), this follows from

$$\mathsf{in}_1{\star}\mathrm{id}_{G_1+G_2} = \mathsf{in}_1 = \mathrm{id}_{G_1}{\star}\mathsf{in}_1, \mathsf{in}_2{\star}\mathrm{id}_{G_1+G_2} = \mathsf{in}_2 = \mathrm{id}_{G_2}{\star}\mathsf{in}_2,$$

For 2), this follows from the equation below (and the symmetric one)

$$\mathsf{in}_1{\star}((m_1{+}m_2){\star}(m'_1{+}m'_2)) = m_1{\star}\mathsf{in}_1{\star}(m'_1{+}m'_2) = (m_1{\star}m'_1){\star}\mathsf{in}_1$$

$\square$

Fact 4.1.16 Suppose m and m' are morphisms from $G_1{+}G_2$ to G', and suppose we have

$$\mathsf{in}_1{\star}m = \mathsf{in}_1{\star}m', \mathsf{in}_2{\star}m = \mathsf{in}_2{\star}m'.$$

Then $m = m'$. $\square$

Observation 4.1.17 Given graph G, it is obviously possible to find a morphism smash from $G{+}G$ to G such that

$$\mathsf{in}_1{\star}\mathsf{smash} = \mathrm{id}_G, \mathsf{in}_2{\star}\mathsf{smash} = \mathrm{id}_G$$

$\square$

Given graphs G_1 and G, we now give a sufficient condition for the existence of a graph G_2 such that there is a specialization from $G_1{+}G_2$ to G:

Lemma 4.1.18 Let G_1 and G be given. Suppose there exists a morphism m from G_1 to G with the following properties:

1. m is respects all active and passive nodes.
2. If $m(a_1) = m(a_2)$, with $a_1 \neq a_2$, then a_1 and a_2 are partial applications.

Then there exists G_2 and specialization s from $G_1{+}G_2$ to G, such that $s(\mathsf{in}_1(n)) = m(n)$ for n in G_1. $\square$

Proof: First we define G_2. There will be four types of nodes: 1) active nodes in G which are not of form $m(a)$ 2) passive nodes in G which are not of form $m(p)$ 3) virtual nodes in G which are not of form $m(v)$ (and hence do not belong to $rg(m)$ 4) some "new" virtual nodes – more about that soon.

The labels of the passive nodes are inherited from G. The children of a node n in G_2 are found as follows: as n also belongs to G, we can talk about its i'th child (i.e. $\mathcal{S}(n,i)$) in G, to be called n'. If n' is of type 1,2 or 3, then let n' be the i'th child of n in G_2 also. Otherwise, create a new virtual node of type 4 and let this node be the i'th child of n in G_2.

Now we have to define s, a mapping from G_1+G_2 to G. Given n in G_1+G_2. If $n = \mathsf{in}_1(n_1)$ with n_1 in G_1, then we let $s(n) = m(n_1)$. If $n = \mathsf{in}_2(n_2)$ with n_2 in G_2 of type 1,2 or 3, we let $s(n) = n_2$ (a node in G). Finally, suppose $n = \mathsf{in}_2(v)$, with v a node in G_2 of type 4. There must exist a unique node n' in G_2, active or passive, such that v is the i'th child of n' in G_2. Then we stipulate that $s(n)$ should be the i'th child of n' in G.

It is rather straightforward to see that this yields a specialization. □

4.1.3 Pushouts

As promised in the introduction of this chapter, we now embark on the following crucial task: to formalize that "if there is a reduction r from g to g', and g is more general than G, then r gives rise to a reduction r' from G to some new graph G', where g' is more general than G'":

Definition 4.1.19 Given a specialization s from g to G, and a reduction r from g to g'. We say that (G',r',s') (or just (r', s')) is a *pushout* of (r,s) iff

- r' is a reduction from G to G'.
- s' is a specialization from g' to G'.
- $s{\star}r' = r{\star}s'$
- If G'' is a graph, s'' is a morphism from g' to G'' and r'' is a morphism from G to G'' [4] such that $s{\star}r'' = r{\star}s''$, then there exists a morphism

[4]Notice that s''/r'' is not necessarily assumed to be a specialization/reduction.

$$\begin{tikzcd}
 & & G'' \\
G \arrow[r, "r'"] \arrow[urr, "r''"] & G' \arrow[ur, "h"] & \\
g \arrow[u, "s"] \arrow[r, "r"] & g' \arrow[u, "s'"] \arrow[uur, "s''"] &
\end{tikzcd}$$

Figure 4.3: The pushout property.

h from G' to G'' such that $r' \star h = r''$, $s' \star h = s''$ – as illustrated in figure 4.3.

□

Fact 4.1.20 Suppose (G', r', s') is a pushout of (r, s). Given r'', s'' as in definition 4.1.19, the h will be unique. Further, if s'' is a specialization (and r'' is a reduction), h will be a homomorphism. □

Proof: The uniqueness of h follows from $s' \star h = s''$ and s' being surjective. Now suppose that r'' is a reduction and s'' is a specialization. We have to check that h respects all passive and active nodes.

Given P' in G'. As s' is a specialization, there exists p' in g' such that $s'(p') = P'$. As s'' is a specialization, $s''(p')$ is passive, i.e. $h(P')$ is passive. Also, $\mathcal{L}''(h(P')) = \mathcal{L}'(P')$. Now let $N' = \mathcal{S}'(P', i)$ for i in the appropriate domain, and let $n' = \mathcal{S}'(p', i)$. Then $s'(n') = N'$, as s' is a specialization. Now, as s'' is a specialization,

$$h(\mathcal{S}'(P', i)) = h(N') = h(s'(n')) = s''(n') = \mathcal{S}''(s''(p'), i) = \mathcal{S}''(h(P'), i)$$

In the same way it can be showed that h respects active nodes. □

Observation 4.1.21 Standard categorical reasoning shows that the pushout, if it exists, is unique up to isomorphism: suppose (G'', r'', s'') as well as (G', r', s') is a pushout of (r, s). Then there exists morphism h such that $r' \star h = r''$, $s' \star h = s''$ and also there exists morphism h' such that $r'' \star h' = r'$, $s'' \star h' = s'$. By fact 4.1.20, h and h' are homomorphisms.

Now $r''\star(h'\star h) = r''$, and $s''\star(h'\star h) = s''$. Still by fact 4.1.20, there exists *unique* morphism h'' such that $r''\star h'' = r''$, $s''\star h'' = s''$. Clearly $h'' = \mathrm{id}_{G''}$, so $h'\star h = \mathrm{id}_{G''}$. Similarly, $h\star h' = \mathrm{id}_{G'}$. Lemma 4.1.8 then tells us that h and h' are isomorphisms. □

Thus the following is well-defined (up to isomorphism):

Definition 4.1.22 Given reduction r from g to g', and specialization s from g to G. If (G', r', s') is a pushout of (r, s), we write $r' = \mathcal{R}_s(r)$, $s' = \mathcal{S}_s(r)$. □

4.1.4 Algebraic laws

Fact 4.1.23 With symbols having the appropriate functionality, we have

1. The pushout of $(r, \mathrm{id\_})$ is $(r, \mathrm{id\_})$, i.e.

$$\mathcal{R}_{\mathrm{id\_}}(r) = r, \mathcal{S}_{\mathrm{id\_}}(r) = \mathrm{id\_} \tag{4.1}$$

2. The pushout of $(\mathrm{id\_}, s)$ is $(\mathrm{id\_}, s)$, i.e.

$$\mathcal{R}_s(\mathrm{id\_}) = \mathrm{id\_}, \mathcal{S}_s(\mathrm{id\_}) = s \tag{4.2}$$

3. Suppose that (r_1, s_1') is a pushout of (r, s_1), and (r_2, s_2') is a pushout of (r_1, s_2). Then $(r_2, s_1'\star s_2')$ is a pushout of $(r, s_1\star s_2)$, i.e.

$$\mathcal{R}_{s_1\star s_2}(r) = \mathcal{R}_{s_2}(\mathcal{R}_{s_1}(r)), \mathcal{S}_{s_1\star s_2}(r) = \mathcal{S}_{s_1}(r)\star\mathcal{S}_{s_2}(\mathcal{R}_{s_1}(r)) \tag{4.3}$$

4. Suppose that (r_1', s_1) is a pushout of (r_1, s), and (r_2', s_2) is a pushout of (r_2, s_1). Then $(r_1'\star r_2', s_2)$ is a pushout of $(r_1\star r_2, s)$, i.e.

$$\mathcal{R}_s(r_1\star r_2) = \mathcal{R}_s(r_1)\star\mathcal{R}_{\mathcal{S}_s(r_1)}(r_2), \mathcal{S}_s(r_1\star r_2) = \mathcal{S}_{\mathcal{S}_s(r_1)}(r_2) \tag{4.4}$$

□

Proof: This is standard categorical reasoning. First we will prove (4.1): clearly $r\star\mathrm{id\_} = \mathrm{id\_}\star r$. Now suppose $r\star s'' = \mathrm{id\_}\star r''$. Then we can choose $h = s''$, as then $r\star h = r''$, $\mathrm{id\_}\star h = s''$.

(4.2) is proven dually. Now for (4.3) (and (4.4) will follow dually), where the situation is as depicted in figure 4.4: clearly $\mathcal{R}_{s_2}(\mathcal{R}_{s_1}(r))$ is a reduction, and $\mathcal{S}_{s_1}(r)\star\mathcal{S}_{s_2}(\mathcal{R}_{s_1}(r))$ is a specialization. Also we have

$$r\star\mathcal{S}_{s_1}(r)\star\mathcal{S}_{s_2}(\mathcal{R}_{s_1}(r)) = s_1\star\mathcal{R}_{s_1}(r)\star\mathcal{S}_{s_2}(\mathcal{R}_{s_1}(r)) = s_1\star s_2\star\mathcal{R}_{s_2}(\mathcal{R}_{s_1}(r))$$

$$\begin{tikzcd}
{} \arrow[r, "s_1"] \arrow[d, "r"'] & {} \arrow[r, "s_2"] \arrow[d, "\mathcal{R}_{s_1}(r)"'] & {} \arrow[d, "\mathcal{R}_{s_2}(\mathcal{R}_{s_1}(r))"'] \arrow[ddr, "r''"] & \\
{} \arrow[r, "\mathcal{S}_{s_1}(r)"] \arrow[drrr, "s''"'] & {} \arrow[r, "\mathcal{S}_{s_2}(\mathcal{R}_{s_1}(r))"] \arrow[drr, "h"] & {} & \\
& & & {}
\end{tikzcd}$$

Figure 4.4: Algebraic law (3).

Next suppose $r{\star}s'' = s_1{\star}s_2{\star}r''$. There exists h such that $\mathcal{S}_{s_1}(r){\star}h = s''$, $\mathcal{R}_{s_1}(r){\star}h = s_2{\star}r''$. Thus there exists h' such that $\mathcal{S}_{s_2}(\mathcal{R}_{s_1}(r)){\star}h' = h$,

$$\mathcal{R}_{s_2}(\mathcal{R}_{s_1}(r)){\star}h' = r''$$

As we also have

$$\mathcal{S}_{s_1}(r){\star}\mathcal{S}_{s_2}(\mathcal{R}_{s_1}(r)){\star}h' = \mathcal{S}_{s_1}(r){\star}h = s''$$

this shows (4.3). □

Fact 4.1.24 Again with symbols having appropriate functionalities, we have (provided the pushouts mentioned on the right hand side exist)

$$\begin{aligned}
\mathcal{R}_{s_1+s_2}(r_1+r_2) &= \mathcal{R}_{s_1}(r_1)+\mathcal{R}_{s_2}(r_2)\\
\mathcal{S}_{s_1+s_2}(r_1+r_2) &= \mathcal{S}_{s_1}(r_1)+\mathcal{S}_{s_2}(r_2)
\end{aligned}$$

□

PROOF: First consider the calculation, where we use fact 4.1.15:

$$\begin{aligned}
& (r_1+r_2){\star}(\mathcal{S}_{s_1}(r_1)+\mathcal{S}_{s_2}(r_2))\\
= \; & (r_1{\star}\mathcal{S}_{s_1}(r_1))+(r_2{\star}\mathcal{S}_{s_2}(r_2)) = (s_1{\star}\mathcal{R}_{s_1}(r_1))+(s_2{\star}\mathcal{R}_{s_2}(r_2))\\
= \; & (s_1+s_2){\star}(\mathcal{R}_{s_1}(r_1)+\mathcal{R}_{s_2}(r_2))
\end{aligned}$$

Next suppose that $(r_1{+}r_2){\star}s'' = (s_1{+}s_2){\star}r''$, with r'' and s'' being morphisms into G''. Our task is to find h (into G'') such that

$$(\mathcal{R}_{s_1}(r_1){+}\mathcal{R}_{s_2}(r_2)){\star}h = r'', (\mathcal{S}_{s_1}(r_1){+}\mathcal{S}_{s_2}(r_2)){\star}h = s'' \tag{4.5}$$

For a start, we have

$$r_1{\star}\mathsf{in}_1{\star}s'' = \mathsf{in}_1{\star}(r_1{+}r_2){\star}s'' = \mathsf{in}_1{\star}(s_1{+}s_2){\star}r'' = s_1{\star}\mathsf{in}_1{\star}r''$$

Thus there exists morphism h_1 into G'' such that

$$\mathcal{R}_{s_1}(r_1){\star}h_1 = \mathsf{in}_1{\star}r'', \mathcal{S}_{s_1}(r_1){\star}h_1 = \mathsf{in}_1{\star}s''$$

Similarly, also there exists morphism h_2 into G'' such that

$$\mathcal{R}_{s_2}(r_2){\star}h_2 = \mathsf{in}_2{\star}r'', \mathcal{S}_{s_2}(r_2){\star}h_2 = \mathsf{in}_2{\star}s''$$

We will define $h = (h_1{+}h_2){\star}\mathsf{smash}$ (recall that smash, a morphism from $G''{+}G''$ into G'', satisfies $\mathsf{in\_}{\star}\mathsf{smash} = \mathrm{id\_}$). Now we have the calculation

$$\begin{aligned}
& (\mathcal{R}_{s_1}(r_1){+}\mathcal{R}_{s_2}(r_2)){\star}(h_1{+}h_2){\star}\mathsf{smash} \\
= \; & ((\mathcal{R}_{s_1}(r_1){\star}h_1){+}(\mathcal{R}_{s_2}(r_2){\star}h_2)){\star}\mathsf{smash} \\
= \; & ((\mathsf{in}_1{\star}r''){+}(\mathsf{in}_2{\star}r'')){\star}\mathsf{smash} = r''
\end{aligned}$$

where the last equality follows from fact 4.1.16 exploiting that

$$\begin{aligned}
\mathsf{in}_1{\star}((\mathsf{in}_1{\star}r''){+}(\mathsf{in}_2{\star}r'')){\star}\mathsf{smash} = (\mathsf{in}_1{\star}r''){\star}\mathsf{in}_1{\star}\mathsf{smash} &= \mathsf{in}_1{\star}r'', \\
\mathsf{in}_2{\star}((\mathsf{in}_1{\star}r''){+}(\mathsf{in}_2{\star}r'')){\star}\mathsf{smash} = (\mathsf{in}_2{\star}r''){\star}\mathsf{in}_2{\star}\mathsf{smash} &= \mathsf{in}_2{\star}r''
\end{aligned}$$

Hence we have shown the first half of (4.5) – the second is shown in a symmetric way. □

Lemma 4.1.25 Suppose (r', s') is a pushout of (r', s'). Suppose r respects a, a an active node in g. Then r' respects $s(a)$. □

PROOF: We have $r'(s(a)) = s'(r(a))$ which is active. And, since r respects a and s,s' respect all active nodes, we have

$$\begin{aligned}
r'(\mathcal{S}(s(a), i)) &= r'(s(\mathcal{S}(a, i))) = s'(r(\mathcal{S}(a, i))) \\
&= s'(\mathcal{S}'(r(a), i)) = \mathcal{S}'(s'(r(a)), i) = \mathcal{S}'(r'(s(a)), i)
\end{aligned}$$

□

4.1.5 Existence of the pushout

Fact 4.1.26 Given specialization s from g to G, and reduction r from g to g'. Then the pushout of (r, s) always exists. $\square$

PROOF: We are going to give a constructive definition of (G', r', s'): to be more precise, we are going to define G' as the quotient of g' wrt. the equivalence relation $\sim$ defined in

Definition 4.1.27 $\sim$, an equivalence relation on g', is the reflexive transitive closure of $\approx$, where $n'_1 \approx n'_2$ iff there exists n_1, n_2 in g such that $r(n_1) = n'_1$, $r(n_2) = n'_2$, $s(n_1) = s(n_2)$. $\square$

In order to facilitate reasoning about $\sim$, we introduce a transition system where

- the configurations are pairs of the form (n, n') with n in g, n' in g' such that $n' = r(n)$.
- the transition relation $\rhd$ is defined as the union of $\rhd_1$, $\rhd_2$ and $\rhd_3$, where
  1. $(n_1, n'_1) \rhd_1 (n_2, n'_2)$ if $n'_1 = n'_2$.
  2. $(n_1, n'_1) \rhd_2 (n_2, n'_2)$ if $n_2 \neq n_1$, $s(n_1) = s(n_2)$.
  3. $(n_1, n'_1) \rhd_3 (n_2, n'_2)$ if there exists n such that $n \neq n_1$, $s(n_1) = s(n)$, $r(n) = n'_2 (= r(n_2))$, and such that $n \neq n_2$.

Lemma 4.1.28 For all n'_1, n'_2 in g', $n'_1 \sim n'_2$ iff $n'_1 = n'_2 \notin rg(r)$ or there exists n_1,n_2 (with $r(n_1) = n'_1$, $r(n_2) = n'_2$) such that $(n_1, n'_1) \rhd^* (n_2, n'_2)$. $\square$

PROOF: If $(n_1, n'_1) \rhd_1 (n_2, n'_2)$ then $n'_1 = n'_2$, and if $(n_1, n'_1) \rhd_2 (n_2, n'_2)$ or $(n_1, n'_1) \rhd_3 (n_2, n'_2)$ then $n'_1 \approx n'_2$. This shows "if".

To show "only if", suppose $n'_1 \sim n'_2$. If $n'_1 \notin rg(r)$, clearly $n'_1 = n'_2$. So assume that $n'_1, n'_2 \in rg(r)$, with $n'_1 \neq n'_2$. There exists $n'_3 \neq n'_1$ such that $n'_1 \approx n'_3$, $n'_3 \sim n'_2$. Hence also $n'_3 \in rg(r)$. Inductively, we can assume that there exists n_{3b}, n_2 such that $(n_{3b}, n'_3) \rhd^* (n_2, n'_2)$. By definition of $\approx$, there exists n_1, n_{3a} such that $(n_1, n'_1) \rhd_2 (n_{3a}, n'_3)$. Hence, perhaps by doing an intermediate $\rhd_1$-transition, $(n_1, n'_1) \rhd^* (n_2, n'_2)$. $\square$

Lemma 4.1.29 Suppose $(n_1, n_1') \rhd^* (n_2, n_2')$. Then there exists (n_3, n_3') and (n_4, n_4') such that

$$(n_1, n_1') \rhd_1^i (n_3, n_3') \rhd_3^* (n_4, n_4') \rhd_2^j (n_2, n_2')$$

where i and j are either 0 or 1. $\square$

PROOF: We have to make several observations, concerning how one can permute the transitions in a sequence:

- a $\rhd_1$-transition, followed by a $\rhd_1$-transition, is equivalent to one $\rhd_1$-transition.
- a $\rhd_2$-transition, followed by a (non-trivial) $\rhd_1$-transition, is equivalent to a $\rhd_3$-transition.
- a $\rhd_3$-transition, followed by a $\rhd_1$-transition, is equivalent to either a $\rhd_2$-transition or a $\rhd_3$-transition.
- a $\rhd_2$-transition, followed by a $\rhd_2$-transition, is equivalent to either a $\rhd_2$-transition or an empty transition.
- a $\rhd_2$-transition, followed by a $\rhd_3$-transition, is equivalent to either a $\rhd_3$-transition or a $\rhd_1$-transition.

Each of these permutations decrease the length of the transition sequence. When no more are applicable, any remaining $\rhd_1$-transitions will be leftmost and any remaining $\rhd_2$-transitions will be rightmost. $\square$

Lemma 4.1.30 Suppose $(n_1, n_1') \rhd_3 (n_2, n_2')$. Suppose that we do not have both n_2 active and n_2' virtual. Then we can draw the following inferences: (we use n for the node in g such that $s(n_1) = s(n)$, $r(n) = n_2'$, $n \neq n_1$, $n \neq n_2$)

- First we observe that n cannot be virtual. For then n_2' will be so too, and as r is injective on virtual nodes and $n \neq n_2$ we will have n_2 active contradicting our assumption.
- Assume n_1 is passive (so also n_1' is passive). As n cannot be virtual, n will be passive. Then n_2' is passive, and n_2' has the same label as n as n_1 as n_1'. Moreover, for i in the appropriate domain we have

$$s(\mathcal{S}(n_1, i)) = \mathcal{S}(s(n_1), i) = \mathcal{S}(s(n), i) = s(\mathcal{S}(n, i))$$

and hence also

$$\mathcal{S}'(n_1', i) = r(\mathcal{S}(n_1, i)) \approx r(\mathcal{S}(n, i)) = \mathcal{S}'(n_2', i)$$

- Assume n_1 is active. As n cannot be virtual and $s(n) = s(n_1)$ which is active, we from s being a specialization and $n \neq n_1$ deduce that n as well as n_1 is a partial application. Hence also n_1' and n_2' are partial applications. As before, for i in the appropriate domain we have

  $$\mathcal{S}'(n_1', i) \approx \mathcal{S}'(n_2', i)$$

$\square$

Lemma 4.1.31 Suppose $(n_1, n_1') \rhd_3^* (n_2, n_2')$. Then we can draw the following inferences:

- If n_1' is active, then either
  - $n_1' = n_2'$
  - n_2' is virtual, n_2 is active
  - n_1' and n_2' are partial applications, and $\mathcal{S}'(n_1', i) \sim \mathcal{S}'(n_2', i)$.
- If n_1' is passive, then either
  - $n_1' = n_2'$
  - n_2' is virtual, n_2 is active
  - n_2' is passive as well, with same label as n_1', and $\mathcal{S}'(n_1', i) \sim \mathcal{S}'(n_2', i)$.
- If n_1' is virtual and n_1 is active, then also n_2' is virtual and n_2 is active.

$\square$

Proof: Induction in the length of the $\rhd_3^*$-sequence. If the length is zero, the claim is trivial. Otherwise, assume that there exists (n_3, n_3') such that $(n_1, n_1') \rhd_3 (n_3, n_3')$ and $(n_3, n_3') \rhd_3^* (n_2, n_2')$.

First suppose n_1' is active, hence also n_1 is active. Now apply lemma 4.1.30: either n_3' is virtual and n_3 is active, in which case the induction hypothesis gives the rest, or n'_1 as well as n'_3 are partial applications with $\mathcal{S}'(n'_1, i) \approx \mathcal{S}'(n'_3, i)$. Also here, the induction hypothesis gives the rest.

Next suppose n'_1 is passive. If n_1 is active, lemma 4.1.30 tells us that (as n_1 cannot be a partial application) n'_3 is virtual and n_3 is active, and then the induction hypothesis gives the rest. So we can assume n_1 to be passive. Then lemma 4.1.30 tells us that either n'_3 is virtual and n_3 is active, in which case the induction hypothesis gives the rest, or n'_3 is a passive node with same label as n'_1 and $\mathcal{S}'(n'_1, i) \approx \mathcal{S}'(n'_3, i)$. Also here, the induction hypothesis gives the rest.

Finally, if n'_1 is virtual and n_1 is active then (as n_1 cannot be a partial application) by lemma 4.1.30 we find that n'_3 is virtual and n_3 is active. Then apply the induction hypothesis. $\square$

Lemma 4.1.32 Suppose $(n_1, n'_1) \rhd_2 (n_2, n'_2)$. Suppose we do not have n'_2 virtual (and thus neither n_2 is virtual). Then we can draw the following inferences (using $s(n_1) = s(n_2)$ and $n_1 \neq n_2$):

- If n_1 is passive, then also n_2 is passive (and hence also n'_1 and n'_2 are passive). Moreover, as above we see that n'_1 and n'_2 have the same label, and that

  $$\mathcal{S}'(n'_1, i) \approx \mathcal{S}'(n'_2, i)$$

- If n_1 is active, then n_1 and n_2 must be partial applications. Hence also n'_1 and n'_2 are partial applications, and as above we have

  $$\mathcal{S}'(n'_1, i) \approx \mathcal{S}'(n'_2, i)$$

$\square$

Now we are ready to prove that $\sim$ is "well-behaved":

Lemma 4.1.33 Suppose $n'_1 \sim n'_2$. Suppose $n'_1 \neq n'_2$, and suppose n'_2 is not virtual. Then

- if n'_1 is active, then n'_2 as well as n'_1 is a partial application, and $\mathcal{S}'(n'_1, i) \sim \mathcal{S}'(n'_2, i)$.
- if n'_1 is passive, then n'_2 is passive with same label as n'_1, and $\mathcal{S}'(n'_1, i) \sim \mathcal{S}'(n'_2, i)$.

□

PROOF: By lemma 4.1.28 and lemma 4.1.29 there exist n_1, n_2 such that (n_2, n'_2) can be derived from (n_1, n'_1) by zero or more $\rhd_3$-steps followed by at most one $\rhd_2$-step. If $(n_1, n'_1) \rhd_3^* (n_2, n'_2)$, the claim follows from lemma 4.1.31. So assume that there exists (n_3, n'_3) such that

$$(n_1, n'_1) \rhd_3^* (n_3, n'_3) \rhd_2 (n_2, n'_2)$$

In the following we can exclude the possibility that n'_3 is virtual and n_3 active, as then lemma 4.1.32 tells us that n'_2 is virtual. Now apply lemma 4.1.31 in the two cases:

- n'_1 is active: then either $n'_1 = n'_3$ or n'_1 and n'_3 are both partial applications with $\mathcal{S}'(n'_1, i) \sim \mathcal{S}'(n'_3, i)$. In both cases, n_3 is active so lemma 4.1.32 tells us (as n'_2 is not virtual) that n'_3 as well as n'_2 are partial applications, with $\mathcal{S}'(n'_2, i) \sim \mathcal{S}'(n'_3, i)$.
- n'_1 is passive: then either $n'_1 = n'_3$ or n'_3 is also passive with same label as n'_1 and with $\mathcal{S}'(n'_1, i) \sim \mathcal{S}'(n'_3, i)$. In both cases, n_3 is either active or passive, and then lemma 4.1.32 tells us (as n'_2 is not virtual, and as n'_3 is not a partial application) that n'_2 is passive with same label as n'_3 and with $\mathcal{S}'(n'_2, i) \sim \mathcal{S}'(n'_3, i)$.

□

Defining G', r', s'

Lemma 4.1.33 shows that the following is well-defined:

Definition 4.1.34 We define G' as follows:

- The nodes of G' are the equivalence classes of the nodes of g' wrt. $\sim$.
- A node in G' will be active iff it (viewed as a set of g'-nodes) contains an active node; a node is passive iff it contains a passive node; consequently a node is virtual iff it contains virtual nodes only.
- If P' is passive in G', $P' = [p']_\sim$ with p' passive in g', then $\mathcal{L}'(P') = \mathcal{L}'(p')$.

- If P' is passive in G', $P' = [p']_\sim$, then for $i \in \{1 \dots Ar(\mathcal{L}(p'))\}$ we define

  $$\mathcal{S}'(P', i) = [\mathcal{S}'(p', i)]_\sim$$

  Similarly, if A' is active with $A' = [a']_\sim$ we define for $i = 1, 2$

  $$\mathcal{S}'(A', i) = [\mathcal{S}'(a', i)]_\sim$$

$\square$

Now we are able to define r' and s':

Definition 4.1.35 s', a morphism from g' to G', is defined by stipulating $s'(n') = [n']_\sim$.

r', a morphism from G to G', is defined as follows: let N be a node in G. Let n be such that $s(n) = N$. Now let $r'(N) = [r(n)]_\sim$. $\square$

We have to ensure that r' is well-defined: assume $s(n_1) = s(n_2) = N$ with $n_1 \neq n_2$. Then $r(n_1) \approx r(n_2)$, hence $[r(n_1)]_\sim = [r(n_2)]_\sim$.

We have to show that $r \star s' = s \star r'$; so consider n in G. Now

$$(s \star r')(n) = r'(s(n)) = [r(n)]_\sim = s'(r(n)) = (r \star s')(n)$$

Next assume s'' is a morphism from g' to G'', and assume r'' is a morphism from G to G'' such that $r \star s'' = s \star r''$. We have to define h, a morphism from G' to G'', such that $r' \star h = r''$, $s' \star h = s''$. So let N' be a node in G'. Find n' in g' such that $[n']_\sim = N'$. Then define $h(N') = s''(n')$.

Of course we have to check this is well-defined, i.e. that $n'_1 \sim n'_2$ implies $s''(n'_1) = s''(n'_2)$. It will be enough to show that $n'_1 \approx n'_2$ implies $s''(n'_1) = s''(n'_2)$. So assume there exists n_1, n_2 with $r(n_1) = n'_1$, $r(n_2) = n'_2$, $s(n_1) = s(n_2)$. Now

$$s''(n'_1) = s''(r(n_1)) = r''(s(n_1)) = r''(s(n_2)) = s''(r(n_2)) = s''(n'_2)$$

By definition, $s'' = s' \star h$. We are left with showing $r'' = r' \star h$. So consider N in G. Let n in g be such that $s(n) = N$. Then

$$\begin{aligned} r''(N) &= r''(s(n)) = s''(r(n)) = h(s'(r(n))) \\ &= h(r'(s(n))) = h(r'(N)) = (r' \star h)(N) \end{aligned}$$

r' is a reduction, s' is a specialization

First we show

Lemma 4.1.36 Suppose $(v, v') \rhd^* (n, n')$, with $s(v) = V$ a virtual node in G. Then n' will be virtual, and if $s(n)$ is virtual then $s(n) = V$. □

PROOF: First we define a predicate P with free variables n, n':

$$P(n, n') \equiv n' \text{ is virtual, } n \text{ is virtual implies } s(n) = V$$

This is an invariant of $\rhd_3$, i.e. $(n_1, n_1') \rhd_3 (n_2, n_2')$, $P(n_1, n_1')$ implies $P(n_2, n_2')$. To see this, suppose there exists n with $s(n_1) = s(n)$, $r(n) = n_2'$, $n \neq n_1$, $n \neq n_2$. We can assume that n_1' is virtual, and n_1 virtual implies $s(n_1) = V$. Two cases:

- If n_1 is active, then – as n_1 is not a partial application – n will be virtual. Then also n_2' is virtual, and n_2 cannot be virtual.
- If n_1 is virtual, by assumption $s(n) = s(n_1) = V$. Hence, n is virtual. So also n_2' is virtual, and n_2 cannot be virtual.

Now assume $(v, v') \rhd^* (n, n')$, with $s(v) = V$. By lemma 4.1.29, the situation is

$$(v, v') \rhd_1^i (n_1, n_1') \rhd_3^* (n_2, n_2') \rhd_2^j (n, n'), i, j \in \{0, 1\}$$

Clearly, $P(n_1, n_1')$ holds. Due to P being an invariant of $\rhd_3$, also $P(n_2, n_2')$ holds. If $j = 0$, the claim is clear. So suppose $j = 1$, i.e. $s(n_2) = s(n)$ with $n \neq n_2$. Two cases:

- n_2 is active. As $r(n_2) = n_2'$ is virtual, n_2 is not a partial application, so n is virtual hence also n' is virtual. On the other hand, $s(n) = s(n_2)$ is not virtual.
- n_2 is virtual. As $P(n_2, n_2')$ holds, $s(n) = s(n_2) = V$. Hence, n' is virtual.

□

Fact 4.1.37 r' is a reduction. □

PROOF: First we show that r' respects passive nodes: let P be a passive node in G. There exists p in g with $s(p) = P$. Now $r(p)$ will be passive, so $[r(p)]_\sim = r'(P)$ is passive. Moreover, $r'(P)$ has the same label as $r(p)$ as p as P. Finally,

$$\begin{aligned} r'(\mathcal{S}(P,i)) &= r'(s(\mathcal{S}(p,i))) = s'(r(\mathcal{S}(p,i))) = [\mathcal{S}(r(p),i)]_\sim \\ &= \mathcal{S}([r(p)]_\sim, i) = \mathcal{S}(s'(r(p)),i) = \mathcal{S}(r'(P),i) \end{aligned}$$

Next we show that r' respects partial applications: let A be a partial application in G. There exists a in g with $s(a) = A$. a will be a partial application. As r respects partial applications, $r(a)$ will be active so $[r(a)]_\sim = r'(A)$ will be active. Moreover, as above we have

$$r'(\mathcal{S}(A,i)) = \mathcal{S}(r'(A),i)$$

We also have to show that r' respects virtual nodes. So let V be virtual in G, and let v be such that $s(v) = V$. Let $v' = r(v)$. We want to show that $[v']_\sim$ is virtual. This can be done by showing that $v' \sim n'$ implies n' is virtual. So assume $v' \sim n'$, $v' \neq n'$. By lemma 4.1.28 there exists n such that $(v, v') \rhd^* (n, n')$. By lemma 4.1.36, n' will be virtual as desired.

Turning to the requirement 2 for a reduction, i.e. that r' is injective on virtual nodes, suppose $r'(V_1) = r'(V_2)$ with v_1, v_2 such that $s(v_1) = V_1$, $s(v_2) = V_2$. With $v_1' = r(v_1)$, $v_2' = r(v_2)$ this means that $v_1' \sim v_2'$. If $v_1' = v_2'$, $v_1 = v_2$ and $V_1 = V_2$. So assume $v_1' \neq v_2'$, then lemma 4.1.28 tells us that there exists n_1, n_2 such that $(n_1, v_1') \rhd^* (n_2, v_2')$. But then also $(v_1, v_1') \rhd^* (v_2, v_2')$. Lemma 4.1.36 now says that $V_1 = V_2$. □

Fact 4.1.38 s' is a specialization. □

PROOF: The only non-trivial point is 2 – but this follows from lemma 4.1.33. □
This concludes the proof of fact 4.1.26. □

Example 4.1.39 Consider the specialization s presented in figure 4.1 and the reduction r_2 presented in figure 4.2. With $r = r_2 + \mathrm{id}_G$ for $G = gv_13$, let us find the pushout of (r, s). This is depicted in figure 4.5, where all application nodes have been numbered for reference purposes.

r is defined as follows: $r(a_1) = v$, $r(a_2) = a_2'$, $r(a_3) = a_3'$, $r(g) = g$, $r(v_1) = v_1$, $r(3) = 3$, $r(v) = v$, $r(f) = f$ (garbage collected in the figure).

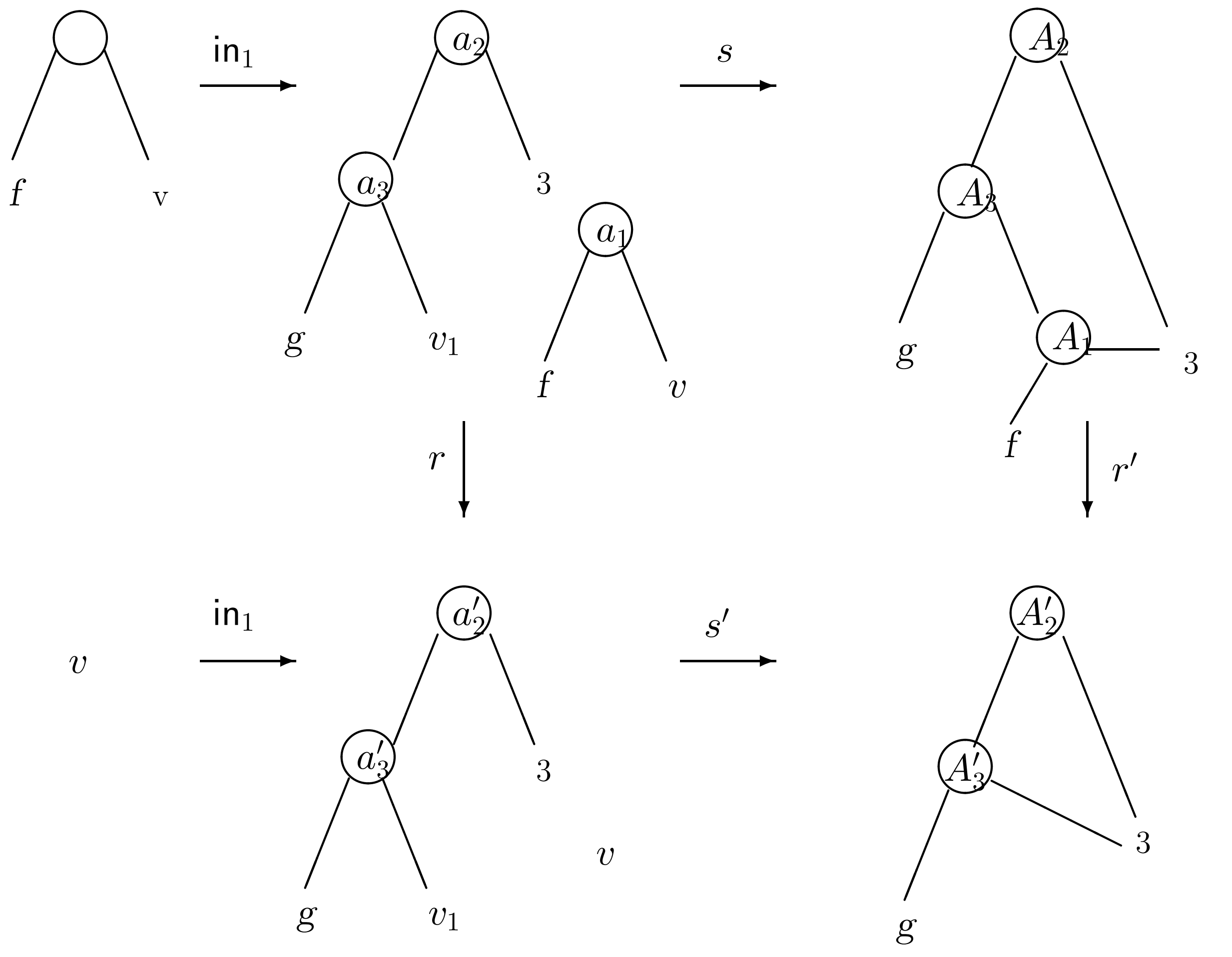


Figure 4.5: A pushout.

s is defined as follows: $s(a_1) = A_1$, $s(a_2) = A_2$, $s(a_3) = A_3$, $s(g) = g$, $s(f) = f$, $s(3) = 3$, $s(v_1) = A_1$, $s(v) = 3$.

We notice that $v \approx v_1$, and $v \approx 3$. Hence the only non-singleton equivalence class is the one containing v, v_1 and 3. This class will be passive.

We denote $A'_2 = [a'_2]_\sim$, and $A'_3 = [a'_3]_\sim$. Then, $r'(A_1) = [r(a_1)]_\sim = 3$, $r'(A_2) = [r(a_2)]_\sim = A'_2$, $r'(A_3) = [r(a_3)]_\sim = A'_3$, $r'(g) = g$, $r'(f) = f$ and $r'(3) = 3$.

Thus, the behavior of the pushout is "as expected". $\square$

Example 4.1.40 Consider the pushout depicted in figure 4.6, where r and r' both map the application node into the virtual node. This pushout may seem a bit "pathological" in the sense that even though r does not "create" any virtual nodes, r' does. On the other hand, since r corresponds to the rule *f(x)* $\rightarrow$x we cannot expect the term *f(f(... (f(f(...)))))*

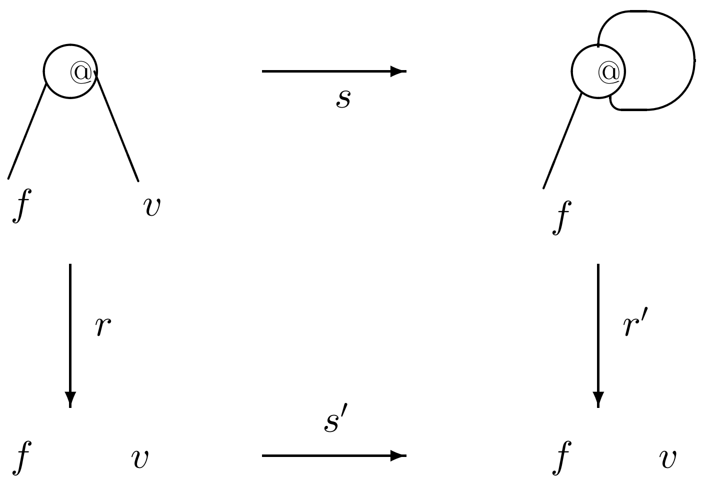


Figure 4.6: A "pathological" pushout.

(which corresponds to the graph in the upper right corner) to have a well-defined value[5]. □

4.1.6 A property of the pushout

Given G and G_L, it may be possible to find several G_i's and s_i's such that s_i is a specialization from $G_L{+}G_i$ to G. The following lemma shows that it doesn't really matter which one we choose, as long as all s_i coincide on G_L:

Lemma 4.1.41 Suppose r is a reduction from G_L to G_R. Suppose s_1 is a specialization from $G_L{+}G_1$ to G and s_2 is a specialization from $G_L{+}G_2$ to G such that for all n in G_L, $s_1(n) = s_2(n)$[6]. Let (G_1', r_1', s_1') be the pushout of $(r{+}\mathrm{id}_{G_1}, s_1)$ and let (G_2', r_2', s_2') be the pushout of $(r{+}\mathrm{id}_{G_2}, s_2)$. The situation is as depicted in figure 4.7. Then there exists an isomorphism h from G_1' to G_2' such that $r_1' \star h = r_2'$. □

PROOF: Let G_1' be the quotient of $G_R{+}G_1$ wrt. $\sim_1$, and let G_2' be the quotient of $G_R{+}G_2$ wrt. $\sim_2$, as in the proof of fact 4.1.26.

We are going to define h as follows: given N' in G_1', find n' in $G_R{+}G_1$ such that $[n']_{\sim_1} = N'$. If $n' \notin rg(r{+}\mathrm{id}_{G_1})$, then n' will belong to G_R and we can prescribe $h(N') = [n']_{\sim_2}$. Otherwise, find n in $G_L{+}G_1$ such that $(r{+}\mathrm{id}_{G_1})(n) = n'$. Now define $h(N') = r_2'(s_1(n))$.

[5]See also section 4.7.5.

[6]We will not bother about writing the injection functions.

$$\begin{tikzcd}
 & G_R+G_1 \arrow[dd, "s'_1"] \\
G_L+G_1 \arrow[ur, "r+\mathrm{id}_{G_1}"'] \arrow[dd, "s_1"'] & \\
 & G'_1 \arrow[dd, "h"] \\
G \arrow[ur, "r'_1"'] \arrow[dr, "r'_2"] & \\
 & G'_2 \\
G_L+G_2 \arrow[uu, "s_2"] \arrow[dr, "r+\mathrm{id}_{G_2}"] & \\
 & G_R+G_2 \arrow[uu, "s'_2"']
\end{tikzcd}$$

Figure 4.7: Two “equivalent” pushouts.

This “definition” involves two choices, so we have to ensure that $h(N')$ actually is independent of those. For the latter choice (how to find n), assume that $n_1 \neq n_2$ but $(r+\mathrm{id}_{G_1})(n_1) = (r+\mathrm{id}_{G_1})(n_2) = n'$. It must hold that n_1, n_2 both belong to G_L, and $r(n_1) = r(n_2) = n'$. By assumption, we have $s_1(n_1) = s_2(n_1)$, $s_1(n_2) = s_2(n_2)$. Hence

$$\begin{aligned} & r'_2(s_1(n_1)) = r'_2(s_2(n_1)) = s'_2((r+\mathrm{id}_{G_1})(n_1)) \\ = \; & s'_2((r+\mathrm{id}_{G_1})(n_2)) = r'_2(s_2(n_2)) = r'_2(s_1(n_2)) \end{aligned}$$

For the former choice (how to find n'), it will be sufficient if we show that if $n'_1 \approx_1 n'_2$, $[n'_1]_{\sim_1} = N'$ $(= [n'_2]_{\sim_1})$ then $h(N')$ does not depend on whether we choose n'_1 or n'_2. There will exist n_1, n_2 such that $(r+\mathrm{id}_{G_1})(n_1) = n'_1$, $(r+\mathrm{id}_{G_1})(n_2) = n'_2$, $s_1(n_1) = s_1(n_2)$. Thus also $r'_2(s_1(n_1)) = r'_2(s_1(n_2))$ as desired.

Notice that the definition of h can be stated in another way: given N' in G'_1, find N in G such that $r'_1(N) = N'$ and let $h(N') = r'_2(N)$. If no such N exists, N' must be a singleton class of form $[n']_{\sim_1}$ with n' in G_R not in $rg(r)$; then define $h(N') = [n']_{\sim_2}$. It is then quite obvious that $r'_1 \star h = r'_2$.

Dually we can define h', a morphism from G'_2 to G'_1. It then follows from the definitions that $h \star h' = \mathrm{id}_{G'_1}$, $h' \star h = \mathrm{id}_{G'_2}$. By lemma 4.1.8, it in order to show that h is an isomorphism is enough to show that h is a homomorphism, i.e. that h respects all passive and active nodes. Let

N' be such a node in G'_1, and let n' in $G_R{+}G_1$ be a node of same kind (active/passive) with $[n']_{\sim_1} = N'$. Now two possibilities:

- n' belongs to G_R. Whatever $n' \in rg(r)$ or not, since $s_1(n) = s_2(n)$ for n in G_L we have $h(N') = [n']_{\sim_2}$ which is of same kind as n' as N' (and with same label). Moreover, as $\mathcal{S}(n', i)$ also belong to G_R,

$$\begin{aligned} & h(\mathcal{S}(N', i)) = h([\mathcal{S}(n', i)]_{\sim_1}) = [\mathcal{S}(n', i)]_{\sim_2} \\ = \;& \mathcal{S}([n']_{\sim_2}, i) = \mathcal{S}(h(N'), i) \end{aligned}$$

- n' belongs to G_1. Now $h(N') = r'_2(s_1(n'))$. It will be enough if we can show that r'_2 respects $s_1(n')$, for then (as $\mathcal{S}(n', i)$ also belongs to G_1)

$$\begin{aligned} & h(\mathcal{S}(N', i)) = h([\mathcal{S}(n', i)]_{\sim_1}) = r'_2(s_1(\mathcal{S}(n', i))) \\ = \;& r'_2(\mathcal{S}(s_1(n'), i)) = \mathcal{S}(r'_2(s_1(n')), i) = \mathcal{S}(h(N'), i) \end{aligned}$$

  Assume, for the sake of contradiction, that r'_2 does not respect $s_1(n')$. Then $s_1(n')$ is active, so n' is active. Let $a \in G_L{+}G_2$ be such that $s_2(a) = s_1(n')$. As r'_2 does not respect $s_2(a)$, lemma 4.1.25 tells us that $(r{+}\mathrm{id}_{G_2})$ does not respect a. Hence a belongs to G_L. Now by our assumptions

$$s_1(a) = s_2(a) = s_1(n')$$

  with $n' \neq a$, n' active and a active but not a partial application. This is a contradiction, as s_1 is a specialization.

$\square$

4.2 Passive nodes carrying multiple labels

For applications, it is desirable to have graphs where passive nodes can have multiple labels – for instance, one might wish to be able to express that some passive node is a positive integer but $\neq 7$. Also, it is desirable to be able to express that the labels of two (passive) nodes are related in some way – for instance that the value of the first node is twice the value of the second node.

We therefore modify the model developed in the previous section by letting $\mathcal{L}$ be a *non-empty set* of labeling functions – in the special case where this set is a singleton, this amounts to the "old" definition. We will require that if $l_1, l_2 \in \mathcal{L}$, then for all p we have $Ar(l_1(p)) = Ar(l_2(p))$ – thus it makes still sense to write $Ar(p)$. If $l(p) = s$ for all $l \in \mathcal{L}$, we shall write $\mathcal{L}(p) = s$ – otherwise, we term p a *multilabeled node.* If $\mathcal{L}$ is a singleton, we say that G is *singlelabeled.*

Now to model that a node p is a positive integer $\neq 7$, we define $\mathcal{L} = \{l_i | i \in \aleph \setminus \{7\}\}$ where $l_i(p) = i$. And to model that the value of p_1 is twice the value of p_2, we define $\mathcal{L} = \{l_i | i \in Z\}$ where $l_i(p_1) = 2 \cdot i$, $l_i(p_2) = i$.

In the rest of this section we indicate how to modify the development of section 4.1 in order to reflect the enhanced model.

First note that the notion of a *genuine* partial application carries through; on the other hand a node a with $Sp(a, i) = p$ and p multilabeled is now considered a (non-genuine) partial application.

In the following let m be a morphism from G to G' (equipped with sets of labeling functions $\mathcal{L}$ and $\mathcal{L}'$). That m respects a node n now (only) means that $m(n)$ is of the same kind as n (and in the case of passive nodes of the same arity), and that $\mathcal{S}'(m(n), i) = m(\mathcal{S}(n, i))$ for $i \in \{1 \ldots Ar(n)\}$. The notion that "the label is respected" is now captured by the following

Definition 4.2.1 For $l \in \mathcal{L}$ and $l' \in \mathcal{L}'$, we say that $l \propto_m l'$ if for all p in G, $m(p)$ is passive and $l'(m(p)) = l(p)$. □

In the case of $\mathcal{L}$ and $\mathcal{L}'$ being singleton sets, this just amounts to the old definition of "m respects all passive nodes".

Observation 4.2.2 Given $l' \in \mathcal{L}'$, there exists at most one $l \in \mathcal{L}$ such that $l \propto_m l'$.

If m has the property that for any passive node p' in G' there exists p in G with $m(p) = p'$, then the converse relation holds – given $l \in \mathcal{L}$, there exists at most one $l' \in \mathcal{L}'$ such that $l \propto_m l'$.

If $l_1 \propto_{m_1} l_2$, and $l_2 \propto_{m_2} l_3$, then $l_1 \propto_{(m_1 \star m_2)} l_3$. □

Two conditions on morphisms turn out to be of interest:

- we say that m satisfies SPEC, provided for all $l' \in \mathcal{L}'$ there exists (unique) $l \in \mathcal{L}$ such that $l \propto_m l'$.
- we say that m satisfies SAME, provided m satisfies SPEC and that for all $l \in \mathcal{L}$ there exists unique $l' \in \mathcal{L}'$ with $l \propto_m l'$.

Fact 4.2.3 If m_1 and m_2 both satisfy SPEC so does $m_1 \star m_2$. If m_1 and m_2 both satisfy SAME so does $m_1 \star m_2$. □

PROOF: The claim concerning SPEC is trivial (using observation 4.2.2). Concerning SAME, existence is obvious – for uniqueness, we must check that if $l_1 \propto_{(m_1 \star m_2)} l_3'$ and $l_1 \propto_{(m_1 \star m_2)} l_3''$ then $l_3' = l_3''$: from m_1 and m_2 satisfying SPEC we find that there exists l_2',l_1',l_2'' and l_1'' such that $l_2' \propto_{m_2} l_3'$, $l_1' \propto_{m_1} l_2'$, $l_2'' \propto_{m_2} l_3''$ and $l_1'' \propto_{m_1} l_2''$. Using observation 4.2.2, we first see that $l_1' \propto_{(m_1 \star m_2)} l_3'$, $l_1'' \propto_{(m_1 \star m_2)} l_3''$ and then see that $l_1' = l_1 = l_1''$. As m_1 and m_2 satisfy SAME, we then first find $l_2' = l_2''$ and then $l_3' = l_3''$, as desired. □
We now redefine the various kinds of morphisms as follows:

Definition 4.2.4 Isomorphisms and reductions must (in addition) satisfy SAME; homomorphisms and specializations must (in addition) satisfy SPEC. □

Observation 4.2.5 If r is a reduction from G to G', and G is singlelabeled, also G' is singlelabeled. □

The intuition is that reductions should neither increase nor decrease the set of possible labels; while a specialization may decrease this set. For instance, if p can be either 7 og 8 then $r(p)$ also can be either 7 or 8 – on the other hand, knowing p to be either 7 or 8 is *more general* than knowing p to be 7.

Lemma 4.1.8 still holds – to see this, observe that $l_1 \propto_{h_1} l_2$ iff $l_2 \propto_{h_2} l_1$ (to show e.g. the "if"-part, note that given p_1 in G_1 we have $h_1(p_1)$ passive with $h_2(h_1(p_1)) = p_1$. As $l_2 \propto_{h_2} l_1$, $l_1(p_1) = l_2(h_1(p_1)))$. So given l_1, there (since h_2 satisfies SPEC) exists unique l_2 with $l_2 \propto_{h_2} l_1$, i.e. there exists unique l_2 with $l_1 \propto_{h_1} l_2$.

Observation 4.1.11 also carries through (here we need the set of labeling functions to be non-empty). From fact 4.2.3 it follows that fact 4.1.12 still holds.

The + operator, revisited

If G_1 and G_2 are equipped with sets of labeling functions $\mathcal{L}_1$ and $\mathcal{L}_2$, we equip $G_1 + G_2$ with a set of labeling function $\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2$ determined by

$$\mathcal{L} = \{l_1 + l_2 | l_1 \in \mathcal{L}_1, l_2 \in \mathcal{L}_2\}$$

where $(l_1 + l_2)(\mathsf{in}_1(p_1)) = l_1(p_1)$, $(l_1 + l_2)(\mathsf{in}_2(p_2)) = l_2(p_2)$.

By observing that $l_1{+}l_2 \propto_{(m_1+m_2)} l_1'{+}l_2'$ iff $l_1 \propto_{m_1} l_1'$ and $l_2 \propto_{m_2} l_2'$, it is easily seen that observation 4.1.13 is still valid. The next interesting point is

Lemma 4.2.6 Lemma 4.1.18 still holds provided we add an extra condition on m: m must satisfy SPEC. $\Box$

Proof: First note that given $l \in \mathcal{L}$, it is possible to find (unique) l_2 such that $l_2 \propto_{s_2} l$ (here s_2 is the restriction of s on G_2). Hence we can define

$$\mathcal{L}_2 = \{l_2 | \exists l \in \mathcal{L} : l_2 \propto_{s_2} l\}$$

Now we are able to show that s satisfies SPEC: given $l \in \mathcal{L}$, there (by assumption) exists $l_1 \in \mathcal{L}_1$ such that $l_1 \propto_{s_1} l$ – and we have just seen that there also exists $l_2 \in \mathcal{L}_2$ such that $l_2 \propto_{s_2} l$. Therefore we have $l_1{+}l_2 \propto_s l$, as desired. $\Box$

Pushouts, revisited

We first show that fact 4.1.20 still holds, i.e. that h satisfies SPEC. So let (with symbols having their obvious meanings) $L'' \in \mathcal{L}''$; our task is to find $L' \in \mathcal{L}'$ such that $L' \propto_h L''$. As r'' satisfies SPEC, there exists (unique) $L \in \mathcal{L}$ such that $L \propto_{r''} L''$. As r' satisfies SAME, there exists unique $L' \in \mathcal{L}$ such that $L \propto_{r'} L'$ (and we want to show that $L' \propto_h L''$). As s satisfies SPEC, there exists (unique) $l \in \mathcal{L}$ such that $l \propto_s L$. As s' satisfies SPEC, there exists (unique) $l_1' \in \mathcal{L}'$ such that $l_1' \propto_{s'} L'$; and as s'' satisfies SPEC there exists (unique) $l_2' \in \mathcal{L}'$ such that $l_2' \propto_{s''} L''$. As r satisfies SPEC, there exists (unique) l_1,l_2 such that $l_1 \propto_r l_1'$, $l_2 \propto_r l_2'$. Now we see (by observation 4.2.2) that

$$l_1 \propto_{(r\star s')} L', l \propto_{(s\star r')} L'$$

and as $r{\star}s' = s{\star}r'$ this (again by observation 4.2.2) implies that $l_1 = l$. In a similar way, we from

$$l_2 \propto_{(r\star s'')} L'', l \propto_{(s\star r'')} L''$$

and $r{\star}s'' = s{\star}r''$ conclude that $l = l_2 (= l_1)$. But since $l \propto_r l_1'$ and $l \propto_r l_2'$ we from r satisfying SAME infer $l_1' = l_2'$.

We are now ready to show that $L' \propto_h L''$: given P' in G', find p' in g' such that $s'(p') = P'$. Then

$$\begin{aligned} L''(h(P')) &= L''(h(s'(p'))) = L''(s''(p')) = l'_2(p') \\ &= l'_1(p') = L'(s'(p')) = L'(P') \end{aligned}$$

Next we turn our attention to fact 4.1.26, i.e. to the *existence* of the pushout. We will present a construct t which to each $L \in \mathcal{L}$ returns a labeling function of G', $t(L)$. Then we can equip G' with the set of labeling functions

$$\mathcal{L}' = \{t(L) | L \in \mathcal{L}\}$$

So let such L be given. As s satisfies SPEC, there exists unique l with $l \propto_s L$. As r satisfies SAME, there exists unique l' with $l \propto_r l'$. Now, for these particular labeling functions l, l' and L we can repeat the development from pp. 55-59 – in particular, we can show that if $p'_1 \sim p'_2$ then $l'(p'_1) = l'(p'_2)$. Hence it makes sense to define $t(L)$ by stipulating that $t(L)(P') = l'(p')$, with p' such that $s'(p') = P'$.

Clearly we have $l' \propto_{s'} t(L)$. Also we have $L \propto_{r'} t(L)$: given P, find p such that $s(p) = P$ – then $r'(P) = s'(r(p))$. Now

$$t(L)(r'(P)) = t(L)(s'(r(p))) = l'(r(p)) = l(p) = L(P).$$

From the above, it is immediate that s' and r' satisfies SPEC. For r' to be a reduction, we must also show that given L there exists unique L' such that $L \propto_{r'} L'$. For existence we can use $L' = t(L)$ – for uniqueness, suppose that we also have $L \propto_{r'} t(L_1)$. As $L_1 \propto_{r'} t(L_1)$ holds, we conclude $L = L_1$.

Finally, we must show that lemma 4.1.41 still holds – it will be enough to show that h is a homomorphism. So let L'_2, a labeling function of G'_2, be given. There exists L_2 (a labeling function of G_2) and L_{R2} (a labeling function of G_R) such that $L_{R2}+L_2 \propto_{s'_2} L'_2$; and L (a labeling function of G) such that $L \propto_{r'_2} L'_2$. Moreover, there exists L_{L2} (a labeling function of G_L) such that $L_{L2} \propto_r L_{R2}$ – then also $L_{L2}+L_2 \propto_{(r+\mathrm{id}_{G_2})} L_{R2}+L_2$. By uniqueness properties, it is now quite easy to see that $L_{L2}+L_2 \propto_{s_2} L$.

Now there exists L'_1 (a labeling function of G'_1) such that $L \propto_{r'_1} L'_1$. Also there exists L_1 and L_{R1} (labeling functions of G_1/G_R) such that $L_{R1}+L_1 \propto_{s'_1} L'_1$. We can find L_{L1} (a labeling function of G_L) such that $L_{L1} \propto_r L_{R1}$ – then also $L_{L1}+L_1 \propto_{r+\mathrm{id}_{G_1}} L_{R1}+L_1$. By uniqueness properties, it is now quite easy to see that $L_{L1}+L_1 \propto_{s_1} L$.

For p in G_L, we now have $L_{L1}(p) = L(s_1(p)) = L(s_2(p)) = L_{L2}(p)$. Hence $L_{L1} = L_{L2}$ – and since r satisfies SAME, also $L_{R1} = L_{R2}$.

We are now in position to show that actually $L'_1 \propto_h L'_2$: let N' in G'_1 be given, and let n' in $G_R{+}G_1$ be such that $s'_1(n') = N'$. We split up the investigation, corresponding to the two cases in the definition of h:

- if $n' \notin rg(r{+}\mathrm{id}_{G_1})$, we have that n' belongs to G_R and $h(N') = s'_2(n')$. Then

$$\begin{aligned} L'_2(h(N')) &= L'_2(s'_2(n')) = L_{R2}(n') \\ &= L_{R1}(n') = L'_1(s'_1(n')) = L'_1(N') \end{aligned}$$

- if n in $G_L{+}G_1$ is such that $(r{+}\mathrm{id}_{G_1})(n) = n'$, we have $h(N') = r'_2(s_1(n))$. Then

$$\begin{aligned} L'_2(h(N')) &= L'_2(r'_2(s_1(n))) = L(s_1(n)) = (L_{L1}{+}L_1)(n) \\ &= (L_{R1}{+}L_1)(n') = L'_1(s'_1(n')) = L'_1(N') \end{aligned}$$

4.3 Modeling demand-driven evaluation

In order to cope with "lazy evaluation", we now extend the model by labeling each node by either 0, 1 or 2. The intuition is as follows:

- If n is labeled 2, n must be reduced to "normal form" – where (loosely speaking) a node is in normal form if it is passive and its children are in normal form.
- If n is labeled 1, n must be reduced to "weak head normal form" (and perhaps later to normal form) – i.e. to a passive node or to a (genuine) partial application.
- If n is labeled 0, n does not (yet) have to be reduced.

We assume the existence of a function Nd, which to each (function) symbol f assigns a "set of needed arguments" $Nd(f)$, where $Nd(f) \subseteq \{1 \ldots Far(f)\}$. The intuition behind $i \in Nd(f)$ is that f cannot be "applied" until its i'th argument has been reduced to "weak head normal form".

Some notation:

- we say that an active node a is *enabled* by f (written $\mathit{En}(a,f)$) if there exists an i and a p such that $\mathit{Sp}(a,i) = p$, $\mathcal{L}(p) = f$ and $\mathit{Far}(f) = i$. Notice that if $\mathit{En}(a,f)$ then $\mathcal{S}(a,i)$ is a genuine partial application (or passive, if $\mathit{Far}(f) = 1$).

- Given active node a, we say that $n = \mathit{Arg}(a,i)$ (n is the i'th argument of a) if either $i = 1$, $n = \mathcal{S}(a,2)$ or there exists active node a' with $a' = \mathcal{S}(a,1)$, $n = \mathit{Arg}(a', i-1)$. Notice that the numbering of the arguments is "reversed".

- Given active node a such that $\mathit{En}(a,f)$. Let $i \in \mathit{Nd}(f)$, and let $n = \mathit{Arg}(a,i)$ (such n will exist). Then we write $\mathit{Ndarg}(a,n)$.

- Given active node a such that $\mathit{En}(a,f)$. Suppose that for all n such that $\mathit{Ndarg}(a,n)$ we have that n is passive. Then we write $\mathit{Rdx}(a)$, formalizing the notion that "a is a redex".

Example 4.3.1 Suppose *f* is defined as below:

f(0,x) $=$ *g(x)*

f(n,x) $=$ *h(x)* *if* $n > 0$

In order to reduce f it is necessary to know the value of its first argument. Accordingly, $\mathit{Nd}(f) = \{2\}$ (remember the ordering of the arguments is reversed!).

Then consider the graph in figure 4.8, where the active nodes (and the passive node with children) have been numbered. Suppose that "initially" node I is labeled 2, i.e. "the context is that" I is to be reduced to normal form. Then also its children are to be reduced to normal form; accordingly we give node II (and the node 12) label 2. We have $\mathit{En}(II,f)$ (but not $\mathit{Rdx}(II)$), and hence (as $IV = \mathit{Arg}(II,2)$) $\mathit{Ndarg}(II,IV)$. Consequently, node IV has to be reduced to weak head normal form, so we give it label 1 (it holds that $\mathit{Rdx}(IV)$). Node VI is not (yet) needed (but we have $\mathit{Rdx}(VI)$), so it will carry label 0. The nodes III, V and VII will carry label 1 (as they already *are* genuine partial applications). □

Now we are ready for

Definition 4.3.2 A *D-graph G* (in the following just called a graph) is a graph together with a function $\mathcal{D}$, the *demand function*, which maps each node into the set {0,1,2}, and which satisfies:

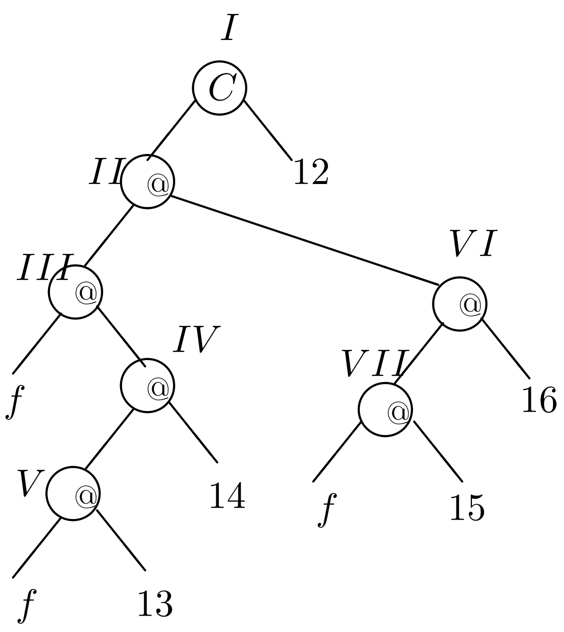


Figure 4.8: A graph illustrating demand-driven evaluation.

1. For all p and for all $i \in \{1 \dots Ar(\mathcal{L}(p))\}$, if $\mathcal{D}(p) = 2$ then $\mathcal{D}(\mathcal{S}(p, i)) = 2$ (if p must be reduced to normal form, so must all its children).
2. For all a, if $\mathcal{D}(a) \geq 1$ then $\mathcal{D}(\mathcal{S}(a, 1)) \geq 1$ (if one has to reduce a to (weak head) normal form, one must know something about its spine).
3. For all a, if $\mathcal{D}(a) \geq 1$ and $Ndarg(a, n)$ then $\mathcal{D}(n) \geq 1$.
4. If n is passive or a genuine partial application, then $\mathcal{D}(n) \geq 1$.

□

Observation 4.3.3 It is easily seen that if $\{\mathcal{D}_i | i \in I\}$ all satisfy 1-4, then so will $\mathcal{D}$ defined by taking pointwise minimum. Also, the $\mathcal{D}$ which maps every node into 2 will satisfy 1-4. Thus, given a mapping D from the nodes of G into $\{0, 1, 2\}$, there exists a least $\mathcal{D}$ such that $\mathcal{D}$ satisfies 1-4 and such that $\mathsf{D} \leq \mathcal{D}$ pointwise. This least $\mathcal{D}$ will be denoted $Clo[G](\mathsf{D})$. □

Definition 4.3.4 Suppose we are given a graph G and a mapping D from the nodes of G into $\{0, 1, 2\}$. Consider the inference system in figure 4.9. Then define

$$\mathcal{D}(n) = \max\{d | \vdash \mathcal{D}(n) \geq d\}$$

□

Due to (4.6) this is well-defined for all n; and due to (4.7) we see that for $d = 0, 1, 2$

$\mathcal{D}(n) \geq d$ iff $\vdash \mathcal{D}(n) \geq d$.

Lemma 4.3.5 For n in G and $d = 0, 1, 2$,

$\mathit{Clo}[G](\mathsf{D})(n) \geq d$ iff $(\vdash)\mathcal{D}(n) \geq d$

and hence also

$\mathit{Clo}[G](\mathsf{D})(n) = \mathcal{D}(n)$

□

PROOF: The "if"-part is a simple induction in the proof tree for $\vdash \mathcal{D}(n) \geq d$. Now for the "only-if"-part: it is easily seen (due to (4.8)-(4.11)) that $\mathcal{D}$ satisfies condition 1-4 and that (due to (4.6)) $\mathsf{D} \leq \mathcal{D}$, and hence (as $\mathit{Clo}[G](\mathsf{D})$ is the least mapping doing so)

$\mathit{Clo}[G](\mathsf{D}) \leq \mathcal{D}$ (pointwise)

by means of which the "only-if"-part follows. □

Let m be a morphism from G to G', equipped with demand functions $\mathcal{D}$ and $\mathcal{D}'$. Several conditions on m turn out to be of interest:

`STEP` m respects all passive nodes; and also respects all active nodes a which do not satisfy $\mathit{Rdx}(a)$.

`EQ` for all n in G, $\mathcal{D}(n) = \mathcal{D}'(m(n))$.

`INC` for all n in G, $\mathcal{D}(n) \leq \mathcal{D}'(m(n))$.

`INC1` there exists function D with $\mathcal{D} = \mathit{Clo}[G](\mathsf{D})$ such that for all n in G, $\mathsf{D}(n) \leq \mathcal{D}'(m(n))$.

`PRES` $\mathcal{D}' = \mathit{Clo}[G'](\mathsf{D}')$, where (with $\max(\emptyset) = 0$)

$$\mathsf{D}'(n') = \max\{\mathcal{D}(n) | m(n) = n'\}$$

`SEQ` m can be written on the form $m = m_1 \star \ldots \star m_n$, with each m_i satisfying `STEP` and `PRES`.

We now state some results concerning the relationship between those conditions:

$$\vdash \mathcal{D}(n) \geq d \qquad \text{if } \mathsf{D}(n) = d \tag{4.6}$$

$$\frac{\vdash \mathcal{D}(n) \geq d}{\vdash \mathcal{D}(n) \geq d'} \qquad \text{if } d > d' \tag{4.7}$$

$$\frac{\vdash \mathcal{D}(p) \geq 2}{\vdash \mathcal{D}(\mathcal{S}(p,i)) \geq 2} \tag{4.8}$$

$$\frac{\vdash \mathcal{D}(a) \geq 1}{\vdash \mathcal{D}(\mathcal{S}(a,1)) \geq 1} \tag{4.9}$$

$$\frac{\vdash \mathcal{D}(a) \geq 1}{\vdash \mathcal{D}(n) \geq 1} \qquad \text{if } \mathit{Ndarg}(a,n) \tag{4.10}$$

$$\vdash \mathcal{D}(n) \geq 1 \qquad \text{if } n \text{ is passive or a genuine partial application} \tag{4.11}$$

Figure 4.9: Inference rules to find $\mathcal{D} = \mathit{Clo}[G](\mathsf{D})$

Observation 4.3.6 If m satisfies `PRES` or `EQ`, then m satisfies `INC`. If m satisfies `INC`, then also m satisfies `INC1`. □

Lemma 4.3.7 If m satisfies `STEP` and `INC1`, then also m satisfies `INC`. □

PROOF: Let $\mathcal{D} = \mathit{Clo}[G](\mathsf{D})$. By lemma 4.3.5, it will be enough to show

$$\vdash \mathcal{D}(n) \geq d \text{ implies } \mathcal{D}'(m(n)) \geq d \tag{4.12}$$

This will be done by induction in the proof tree for the left hand side:

- If (4.6) has been applied, (4.12) follows from m satisfying `INC1`.
- The case where (4.7) has been applied is straightforward.
- Suppose (4.8) has been applied, i.e. we have $\vdash \mathcal{D}(n) \geq 2$ because $n = \mathcal{S}(p,i), \vdash \mathcal{D}(p) \geq 2$. By induction, $\mathcal{D}'(m(p)) \geq 2$. As m satisfies `STEP`, m respects p – hence $m(p)$ is passive and $\mathcal{S}'(m(p),i) = m(n)$. Hence $\mathcal{D}'(m(n)) \geq 2$, as desired.
- Suppose (4.9) has been applied, i.e. we have $\vdash \mathcal{D}(a) \geq 1$ because $n = \mathcal{S}(a,1)$, $\vdash \mathcal{D}(a) \geq 1$. By induction, $\mathcal{D}'(m(a)) \geq 1$. As m satisfies `STEP`, two possibilities:

  - *Rdx*(a) does not hold. Then m respects a, so $m(a)$ is active and $\mathcal{S}'(m(a), 1) = m(n)$. Hence $\mathcal{D}'(m(n)) \geq 1$, as desired.
  - *Rdx*(a) does hold. Then n is a genuine partial application or passive, so $m(n)$ will be a genuine partial application or passive – in both cases, $\mathcal{D}'(m(n)) \geq 1$.

- Suppose (4.10) has been applied, i.e. we have $\vdash \mathcal{D}(n) \geq 1$ because *Ndarg*(a, n), $\vdash \mathcal{D}(a) \geq 1$. By induction, $\mathcal{D}'(m(a)) \geq 1$. Now two possibilities:
  - m respects a. Then, in the usual way, we get $\mathcal{D}'(m(n)) \geq 1$.
  - m does not respect a. Then it must hold that *Rdx*(a), implying that n is passive. But then also $m(n)$ is passive, so $\mathcal{D}'(m(n)) \geq 1$.

- Suppose (4.11) has been applied, i.e. we have $\vdash \mathcal{D}(n) \geq 1$ because n is passive or a genuine partial application. Then, as m satisfies `STEP`, also $m(n)$ is passive or a genuine partial application, hence $\mathcal{D}'(m(n)) \geq 1$.

□

Lemma 4.3.8 Suppose m_1 (from G_1 to G_2) satisfies `PRES`, and suppose m_2 (from G_2 to G_3) satisfies `PRES` as well as `STEP`. Then $m = m_1 \star m_2$ will satisfy `PRES`. □

Proof: From m_1 and m_2 satisfying `PRES` we have the following definitions, where G_1 is equipped with $\mathcal{D}_1$ etc.:

$$\mathsf{D}_2(n_2) = \max\{\mathcal{D}_1(n_1) | m_1(n_1) = n_2\} \quad \text{and} \quad \mathcal{D}_2 = \mathit{Clo}[G_2](\mathsf{D}_2)$$
$$\mathsf{D}_3(n_3) = \max\{\mathcal{D}_2(n_2) | m_2(n_2) = n_3\} \quad \text{and} \quad \mathcal{D}_3 = \mathit{Clo}[G_3](\mathsf{D}_3)$$

Our task is to show that

$$\mathcal{D}_3 = \mathcal{D}_3' \tag{4.13}$$

where we (not quite consistent with our notational conventions) have

$$\mathsf{D}_3'(n_3) = \max\{\mathcal{D}_1(n_1) | m(n_1) = n_3\} \quad \text{and} \quad \mathcal{D}_3' = \mathit{Clo}[G_3](\mathsf{D}_3')$$

First the "$\geq$"-part (where we do not need to assume that m_2 satisfies `STEP`): this can be carried out by showing that for all $n_3 \in G_3$

$$\mathsf{D}_3'(n_3) \leq \mathcal{D}_3(n_3)$$

which in turn can be done by showing that for all n_1 such that $m(n_1) = n_3$

$$\mathcal{D}_1(n_1) \leq \mathcal{D}_3(n_3)$$

But for such n_1 we have

$$\begin{aligned}\mathcal{D}_1(n_1) &\leq \mathsf{D}_2(m_1(n_1)) \leq \mathcal{D}_2(m_1(n_1)) \leq \mathsf{D}_3(m_2(m_1(n_1))) \\ &\leq \mathcal{D}_3(m_2(m_1(n_1))) = \mathcal{D}_3(n_3)\end{aligned}$$

Next the "$\leq$"-part of (4.13). This can be carried out by showing that for all $n_3 \in N_3$

$$\mathsf{D}_3(n_3) \leq \mathcal{D}'_3(n_3)$$

which in turn can be done by showing that for all n_2 such that $m_2(n_2) = n_3$

$$\mathcal{D}_2(n_2) \leq \mathcal{D}'_3(n_3)$$

As m_2 satisfies `STEP`, lemma 4.3.7 tells us it will be sufficient to show that for all $n_2 \in N_2$

$$\mathsf{D}_2(n_2) \leq \mathcal{D}'_3(m_2(n_2))$$

which in turn can be done by showing that for all n_1 such that $m_1(n_1) = n_2$

$$\mathcal{D}_1(n_1) \leq \mathcal{D}'_3(m_2(n_2))$$

But for such n_1 we have

$$\begin{aligned}\mathcal{D}_1(n_1) &\leq \mathsf{D}'_3(m(n_1)) = \mathsf{D}'_3(m_2(n_2)) \\ &\leq \mathcal{D}'_3(m_2(n_2))\end{aligned}$$

□

Lemma 4.3.9 Suppose m satisfies `SEQ`. Then m also satisfies `PRES`. □

PROOF: Induction in the "length" of m, i.e. in the minimal n such that $m = m_1 \star \ldots \star m_n$ with each m_i satisfying `PRES` and `STEP`. If this length is zero, $m = \mathrm{id}\_$ and the claim is clear. Otherwise, we can write $m = m_1 \star m_2$ with m_2 satisfying `PRES` and `STEP`, and with (by the induction hypothesis) m_1 satisfying `PRES`. Lemma 4.3.8 now enables us to conclude that m satisfies `PRES`. □

Lemma 4.3.10 Let m be a morphism from G' to G'', m' be a morphism from G to G' and m'' be a morphism from G to G'', such that $m' \star m = m''$. Suppose m' and m'' satisfies `PRES`, and suppose m satisfies `STEP`. Then m satisfies `INC`. □

PROOF: Let G, G', G'' be equipped with demand functions $\mathcal{D}$, $\mathcal{D}'$ and $\mathcal{D}''$ respectively. We have $\mathcal{D}' = Clo[G'](\mathsf{D}')$ and $\mathcal{D}'' = Clo[G''](\mathsf{D}'')$, where

$$\mathsf{D}'(n') = \max\{\mathcal{D}(n) | m'(n) = n'\}, \mathsf{D}''(n'') = \max\{\mathcal{D}(n) | m''(n) = n''\}$$

Due to lemma 4.3.7, it will be enough to show that $\mathsf{D}'(n') \leq \mathcal{D}''(m(n'))$ for all n' in G'. This amounts to showing that $\mathcal{D}(n) \leq \mathcal{D}''(m(n'))$ for $n' = m'(n)$, i.e. that $\mathcal{D}(n) \leq \mathcal{D}''(m''(n))$ for all n in G. But this is trivial. □

Specializations and reductions, revisited

We now redefine the various kinds of morphisms as follows:

Definition 4.3.11 For D-graphs, we demand

- Isomorphisms to satisfy `EQ`.
- Specializations and homomorphisms to satisfy `INC`.
- Reductions to satisfy `SEQ`.

□

The motivation for demanding specializations to satisfy `INC` is that the existence of a specialization s from g to G should model that g is more general than G - and it is more general for a node to be labeled 0 than to be labeled 1 (than to be labeled 2), as the former denotes that we cannot say anything definite while the latter denotes that we *know* that the node must be reduced to weak head normal form.

Lemma 4.1.8 and fact 4.1.12 still hold with our new definitions.

Concerning +, the only non-trivial point is to verify that $m_1 + m_2$ is a reduction when m_1 and m_2 are: by fact 4.1.15, we have

$$m_1 + m_2 = (m_1 \star \mathrm{id}\_) + (\mathrm{id}\_ \star m_2) = (m_1 + \mathrm{id}\_) \star (\mathrm{id}\_ + m_2)$$

and thus it will be enough to show that $m + \mathrm{id}\_$ is a reduction when m is. We can write $m = m_1 \star \ldots \star m_n$, with each m_i satisfying `STEP` and `PRES`, and proceed by induction in (the minimal such) n. If $n = 0$, $m = \text{id\_}$ and then $\text{id\_}+\text{id\_} = \text{id\_}$ is a reduction. Otherwise, we have $m = m_1 \star m_2$ with m_1 a "shorter" reduction than m and with m_2 satisfying `STEP` and `PRES`. By the induction hypothesis, $m_1+\text{id\_}$ is a reduction; and it is easily seen that $m_2+\text{id\_}$ will satisfy `STEP` and `PRES`. But then

$$(m+\text{id\_}) = (m_1 \star m_2)+(\text{id\_} \star \text{id\_}) = (m_1+\text{id\_}) \star (m_2+\text{id\_})$$

will be a reduction.

Lemma 4.3.12 Lemma 4.1.18 (modified into lemma 4.2.6) still holds provided we add an extra condition on m: it must satisfy `INC`. □

PROOF: The proof is modified as follows: we equip G_2 with a demand function $\mathcal{D}_2 = \mathit{Clo}[G_2](0)$, where 0 is the constant function. We have to check that s satisfies `INC` – but as s satisfies `STEP`, it by lemma 4.3.7 will be enough to show that s satisfies `INC1`. And this can be done by showing that (with symbols having their obvious meaning)

1. $\mathcal{D}_1(n) \leq \mathcal{D}(s(\mathsf{in}_1(n)))$ for all n in G_1
2. $0 \leq \mathcal{D}(s(\mathsf{in}_2(n)))$ for all n in G_2.

But as $s(\mathsf{in}_1(n)) = m(n)$, 1 follows from the new condition imposed. □

Pushouts, revisited

We have to check that what has been said about pushouts is still valid. Concerning the development in section 4.1.3, the only interesting task is to ensure that the second claim of fact 4.1.20 still holds, i.e. that h satisfies `INC`. But as reductions (cf. lemma 4.3.9) satisfy `PRES`, this is a consequence of lemma 4.3.10 (since we know h respects all passive and active nodes).

Concerning the existence of the pushout of a reduction r and a specialization s, we proceed by induction on n where n is the minimal n such that r can be written as $r_1 \star \ldots \star r_n$ with each r_i satisfying `PRES` and `STEP`. Fact 4.1.23,2 caters for the basic step; and fact 4.1.23,4 will take care of the induction step provided we can show that the pushout exists when r satisfies `STEP` and `PRES`: for this purpose let g, g' and G be equipped with demand functions $\mathcal{D}_g$, $\mathcal{D}_{g'}$ and $\mathcal{D}$ respectively. We have $\mathcal{D}_{g'} = \mathit{Clo}[g'](\mathsf{D}_{g'})$, where

$$\mathsf{D}_{g'}(n') = \max\{\mathcal{D}_g(n) | r(n) = n'\}$$

Let (G', r', s') be the pushout (in the "old sense") of (r, s). Now equip G' with demand function $\mathcal{D}' = \mathit{Clo}[G'](\mathsf{D}')$, where

$$\mathsf{D}'(N') = \max\{\mathcal{D}(N) | r'(N) = N'\}$$

The following points must be shown:

- That r' satisfies `PRES`. But this is an immediate consequence of the definition of G'.
- That r' satisfies `STEP`. But suppose A in G is not a redex. Then there exists a in g with $s(a) = A$, such that a is not a redex. Therefore r will respect a. By lemma 4.1.25 we conclude that r' respects A.
- That s' satisfies `INC`, i.e. that for all n' in g' we have $\mathcal{D}_{g'}(n') \leq \mathcal{D}'(s'(n'))$. As s' satisfies `STEP`, lemma 4.3.7 tells us that it is enough to show that $\mathsf{D}_{g'}(n') \leq \mathcal{D}'(s'(n'))$. But this amounts to showing that $\mathcal{D}_g(n) \leq \mathcal{D}'(s'(n'))$ for all n such that $r(n) = n'$, i.e. (as $r \star s' = s \star r'$) that $\mathcal{D}_g(n) \leq \mathcal{D}'(r'(s(n)))$ for all n in g. But for such n we have (as s satisfies `INC`)

  $$\mathcal{D}_g(n) \leq \mathcal{D}(s(n)) \leq \mathsf{D}'(r'(s(n))) \leq \mathcal{D}'(r'(s(n)))$$

That lemma 4.1.41 carries through is a consequence of lemma 4.3.10 (as all we have to check is that h is a homomorphism).

The following lemma expresses that if we only reduce redices the evaluation of which is not demanded, and the resulting graph contains a redex the evaluation of which *is* demanded, then this redex was present already in the original graph.

Lemma 4.3.13 Let G and G' be graphs, equipped with demand functions $\mathcal{D}$ and $\mathcal{D}'$. Suppose r is a reduction from G to G', which respects any node n except if n is a redex and $\mathcal{D}(n) = 0$. Suppose G' contains a redex a' with $\mathcal{D}'(a') \geq 1$. Then there exists a in G with $r(a) = a'$, such that a is a redex in G with $\mathcal{D}(a) \geq 1$. □

PROOF: In the following it is helpful to note that if a is active with $\mathcal{D}(a) \geq 1$ then "the spine and the ribs of a is preserved", e.g. if $r(a)$ is a redex in G' then a is a redex in G, and if $Ndarg'(r(a), n')$ then there exists n with $r(n) = n'$ such that $Ndarg(a, n)$.

By the remark above it will be sufficient to show

1. If n' is active in G', not a partial application and $\mathcal{D}'(n') \geq 1$, then there exists n in G with $\mathcal{D}(n) \geq 1$ such that $r(n) = n'$.
2. If n' is passive in G' and $\mathcal{D}'(n') \geq 2$, then there exists n in G with $\mathcal{D}(n) \geq 2$ such that $r(n) = n'$.

As r satisfies `PRES`, we have that $\mathcal{D}' = Clo[G'](\mathsf{D}')$ where

$$\mathsf{D}'(n') = \max\{\mathcal{D}(n) | r(n) = n'\}$$

We will proceed by induction in the proof tree for $\vdash \mathcal{D}'(n') \geq d$ (where all inferences will have $d \geq 1$):

- axiom 4.6 has been applied, i.e. $\mathsf{D}'(n') \geq d$. Thus there exists n in G with $\mathcal{D}(n) \geq d$ such that $r(n) = n'$, i.e. the claim.
- rule 4.7 has been applied, i.e. $\vdash \mathcal{D}'(n') \geq d'$ with $d' > d$. By induction there exists n in G with $\mathcal{D}(n) \geq d'$ ($> d$) such that $r(n) = n'$.
- rule 4.8 has been applied, i.e. $d = 2$ and $n' = \mathcal{S}'(p', i)$ with $\vdash \mathcal{D}'(p') \geq 2$. By induction, there exists n in G with $\mathcal{D}(n) \geq 2$ such that $r(n) = p'$. By assumption, r respects n, so n is passive and $r(\mathcal{S}(n, i)) = n'$. Moreover $\mathcal{D}(\mathcal{S}(n, i)) \geq 2$.
- rule 4.9 has been applied, i.e. $d = 1$ and $n' = \mathcal{S}'(a', 1)$ with $\vdash \mathcal{D}'(a') \geq 1$. By induction, there exists n in G with $\mathcal{D}(n) \geq 1$ such that $r(n) = a'$. By assumption, r respects n, so n is active and $r(\mathcal{S}(n, i)) = n'$. Moreover $\mathcal{D}(\mathcal{S}(n, i)) \geq 1$.
- rule 4.10 has been applied, i.e. $d = 1$ and $Ndarg(a', n')$ with $\vdash \mathcal{D}'(a') \geq 1$. By induction, there exists n_1 in G with $\mathcal{D}(n_1) \geq 1$ such that $r(n_1) = a'$. By assumption, r respects n_1, implying that n_1 is active. The initial remark tells us that there exists n with $r(n) = n'$ such that $Ndarg(n_1, n)$. Hence $\mathcal{D}(n) \geq 1$.
- axiom 4.11 has been applied, i.e. $d = 1$ and n' is passive or a genuine partial application. Then the claim follows vacuously.

□

4.3.1 Result node

Given node $n0$ in a graph G, let D_{n0} denote the function defined by $\mathsf{D}_{n0}(n0) = 2$, $\mathsf{D}_{n0}(n) = 0$ for $n \neq n0$.

Given graph G equipped with demand function $\mathcal{D}$, we say that $n0$ is a *result node* of G if $\mathcal{D} = \mathit{Clo}[G](\mathsf{D}_{n0})$. The intuition is that the "result of the computation" will be the graph "headed by $n0$", hence $n0$ must be reduced to normal form. Notice that if G is acyclic, G may contain at most one result node.

Fact 4.3.14 Suppose $n0$ is a result node of G, and let r be a reduction from G to G'. Then $r(n0)$ is a result node of G'. □

Proof: We have $r = r_1 \star \ldots \star r_n$ with each r_i satisfying `STEP` and `PRES`. Clearly, it will be sufficient if we can show the claim for each r_i, so in the following we can assume r to satisfy `STEP` and `PRES`.

Let G and G' be equipped with demand functions $\mathcal{D}$ and $\mathcal{D}'$, where $\mathcal{D} = \mathit{Clo}[G](\mathsf{D}_{n0})$ and where (as r satisfies `PRES`) $\mathcal{D}' = \mathit{Clo}[G'](\mathsf{D}')$ with $\mathsf{D}'(n') = \max\{\mathcal{D}(n) | r(n) = n'\}$. Let $\mathcal{D}'_{r(n0)} = \mathit{Clo}[G'](\mathsf{D}'_{r(n0)})$, then our task is to show that $\mathcal{D}' = \mathcal{D}'_{r(n0)}$.

We clearly have $\mathsf{D}_{n0}(n) \leq \mathcal{D}'_{r(n0)}(r(n))$ for all n in G, so as r satisfies `STEP` lemma 4.3.7 tells us that $\mathcal{D}(n) \leq \mathcal{D}'_{r(n0)}(r(n))$ for all n. Hence, $\mathsf{D}' \leq \mathcal{D}'_{r(n0)}$ pointwise and thus also $\mathcal{D}' \leq \mathcal{D}'_{r(n0)}$.

To show the opposite inequality it is sufficient to show that $\mathsf{D}'_{r(n0)} \leq \mathcal{D}'$ pointwise, and this amounts to showing that $\mathcal{D}'(r(n0)) = 2$. But this is immediate. □

4.4 Transitions at level 1

After this long preparation, we are now in position to define what a multilevel transition system (cf. chapter 2) means in a functional framework (i.e. the graph reduction model developed in section 4.1-4.3). In this section we solely focus on the level 1 transitions, i.e. the "normal mode of computation". We shall write

$(r : G \Rightarrow_a G') \in \mathcal{R}_0$

to denote that r, a reduction from G to G' "which reduces the redex a", is a level 0 rule. We shall assume that the level 0 rules are indexed by some set J,[7] that is there exists a J-indexed family $\{(r_j, GL_j, GR_j, a_j) | j \in J\}$ with the property that $(r : G \Rightarrow_a G') \in \mathcal{R}_0$ iff there exists $j \in J$ with $r = r_j$, $G = GL_j$, $G' = GR_j$ and $a = a_j$. Also, we shall write

$$1 \vdash (j)\ r : G \Rightarrow_a G'$$

to denote that r, a reduction from G to G', is a "single step transition" at level 1 which "reduces a by means of the j'th level 0 rule", a a redex in G. Finally, we shall write

$$1 \vdash^* r : G \Rightarrow^c_{Nn} G'$$

to denote that r, a reduction from G to G', consists of c steps the n of which are "normal order steps". It is rather straight-forward how to formalize this; we have the following inference rules (where $\mathcal{D}$ is the demand function of G):

$$\frac{1 \vdash (j)\ r : G \Rightarrow_a G'}{1 \vdash^* r : G \Rightarrow^1_{N1} G'},\ \text{if } \mathcal{D}(a) \geq 1$$

$$\frac{1 \vdash (j)\ r : G \Rightarrow_a G'}{1 \vdash^* r : G \Rightarrow^1_{N0} G'},\ \text{if } \mathcal{D}(a) = 0$$

$$1 \vdash^* \mathrm{id}_G : G \Rightarrow^0_{N0} G$$

$$\frac{1 \vdash^* r_1 : G_1 \Rightarrow^{c_1}_{Nn_1} G_2, 1 \vdash^* r_2 : G_2 \Rightarrow^{c_2}_{Nn_2} G_3}{1 \vdash^* r_1 \star r_2 : G_1 \Rightarrow^{(c_1+c_2)}_{N(n_1+n_2)} G_3}$$

If $1 \vdash^* r : G \Rightarrow^1_{N1} G'$ we say that r is a *normal order step*; if $1 \vdash^* r : G \Rightarrow^1_{N0} G'$ we say that r is a *non-normal order step.* If $1 \vdash^* r : G \Rightarrow^n_{Nn} G'$ for some n, we say that r is a *normal order reduction.*

We are left with defining when $1 \vdash (j)\ r : G \Rightarrow_a G'$ holds:

Definition 4.4.1 Let $(r_j : GL_j \Rightarrow_{a_j} GR_j) \in \mathcal{R}_0$. It will now hold that $1 \vdash (j)\ r : G \Rightarrow_a G'$ provided there exists graph G_1 and specialization s from $GL_j + G_1$ to G such that $(G', r, \_)$ is the pushout of $(r_j + \mathrm{id}_{G_1}, s)$, and such that $a = s(\mathsf{in}_1(a_j))$. $\square$

[7] J is not necessarily assumed to be finite, but of course any given application will use a finite number only.

$$\begin{tikzcd}
GL_j+G_1 \arrow[r, "r_j+\mathrm{id}_{G_1}"] \arrow[d, "s"'] & GR_j+G_1 \arrow[d] \\
G \arrow[r, "r"] & G'
\end{tikzcd}$$

Figure 4.10: Condition for $1 \vdash (j)\ r : G \Rightarrow_a G'$

The situation is depicted in figure 4.10.

The level 0 rules

In order for e.g. the Church-Rosser property to hold, we need to make some assumptions about the level 0 rules:

1. For all $j \in J$, a_j is a redex in GL_j and all other active nodes in GL_j are genuine partial applications (hence r_j will satisfy `STEP`).

   Moreover, for all n in GL_j we have $n \preceq a_j$ (cf. definition 4.1.2) – i.e. a_j is "above" all other nodes.

2. Let G be a graph with redex a. Then at most one rule "matches a", i.e. there exists at most one $j \in J$ such that there exists G_1 and specialization s from GL_j+G_1 to G with $s(\mathsf{in}_1(a_j)) = a$. Moreover, if G is singlelabeled then there must always exist such j.

We will now by means of two examples show that these assumptions are quite natural and thus do not limit the scope of the theory significantly. First let us see how an "operator" could be encoded as a set of level 0 rules; consider + (we have $\mathit{Far}(+) = 2$ and $\mathit{Nd}(+) = \{1, 2\}$ as + needs its both arguments). Then there will exist a single level 0 rule[8] for +, as depicted in figure 4.11 (where no nodes are garbage collected). The left hand side is equipped with a set of labeling functions $\mathcal{L}$, given by

$$\mathcal{L} = \{L_{v1,v2} | v1, v2 \in \mathsf{S}\}$$

where $L_{v1,v2}(p_1) = v1$, $L_{v1,v2}(p_2) = v2$. The right hand side is equipped with a set of labeling functions $\mathcal{L}'$, given by

$$\mathcal{L}' = \{L'_{v1,v2} | v1, v2 \in \mathsf{S}\}$$

[8]This is not quite correct; there should also be ("error") rules where the passive nodes have arity > 0.

We always have $L'_{v1,v2}(p_1) = v1$ and $L'_{v1,v2}(p_2) = v2$ (thus the reduction satisfies SAME). Concerning $L'_{v1,v2}(p)$ we have:

1. $L'_{v1,v2}(p) = v1 + v2$, if $v1$ and $v2$ both are integers;
2. $L'_{v1,v2}(p) = \mathsf{Error}$ (a special element of S) otherwise.

Finally, the left hand sides of the rules are equipped with demand function $Clo[\_](0)$, where 0 is the constant function.

We will now argue that condition 2 is satisfied – uniqueness is trivial, so let us see that if G is singlelabeled and contains redex a and $\mathcal{L}(Sp(a, 2)) = +$ then the level 0 rule for + matches a. Assume wlog. that – with $n_1 = Arg(a, 1)$ and $n_2 = Arg(a, 2)$ – $\mathcal{L}(n_1) = 8$, $\mathcal{L}(n_2) = 7$. With G' the left hand side of figure 4.11 (with redex a'), we want to apply lemma 4.3.12 – to this end we have to find a morphism m from G' to G such that $m(a') = a$ and such that m

1. respects all active and passive nodes;
2. satisfies that if $m(a_1) = m(a_2)$ with $a_1 \neq a_2$ then a_1 and a_2 are partial applications;
3. satisfies `INC`;
4. satisfies SPEC.

m is defined "the natural way" – that 1 holds is clear. For 2, we have to ensure that it cannot hold that $m(a') = m(\mathcal{S}(a', 1))$. But if this was the case, then $a = \mathcal{S}(a, 1)$ and a would not be a redex (as then also $Sp(a, 2) = a$). 3 follows since passive nodes/genuine partial applications (the only nodes with $\mathcal{D}'(n) \geq 1$) are mapped into passive nodes/genuine partial applications. Finally, 4 is obvious – use $L_{7,8}$.

Let us also see how a "user defined function" is encoded as a set of level 0 rules; consider *f* defined by

f([ ],y) = [ ]

f((n :: x),y) = h(x,y)

This gives rise to the two level 0 rules (in addition to an "error rule", if the first argument to *f* is e.g. a number) depicted in figure 4.12 (with some nodes garbage collected).

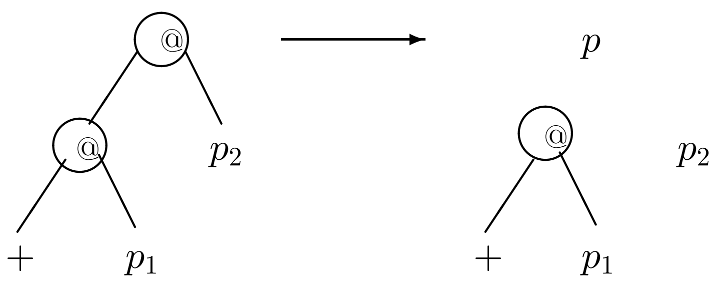


Figure 4.11: Level 0 rules for +

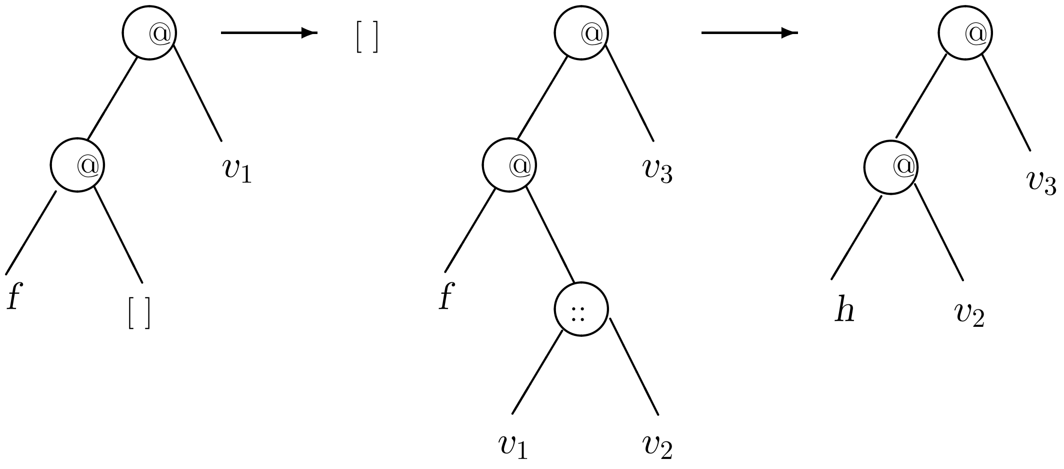


Figure 4.12: Level 0 rules for f

Sometimes one has a choice about how to represent a user defined function: consider e.g. g defined by $g(x) = h(x{+}1)$. Then one can *either* represent the + operator explicitly (as a node in the graph); then $Nd(g) = \emptyset$ *or* one can "code the addition into the labeling functions" (in effect unfold the + operator); then $Nd(g) = \{1\}$. So if h does not need its argument, the former coding should be employed – otherwise some extra strictness is imposed.

The Church-Rosser property

The following lemma shows that "reduction is uniquely determined by the redex being reduced":

Lemma 4.4.2 Suppose $1 \vdash (j1)\ r_1 : G \Rightarrow_a G_1'$ and $1 \vdash (j2)\ r_2 : G \Rightarrow_a G_2'$. Then $j1 = j2$, $r_1 = r_2$ and $G_1' = G_2'$ (modulo isomorphism). □

Proof: Let $(r_{j1} : GL_{j1} \Rightarrow_{a_{j1}} GR_{j1}) \in \mathcal{R}_0$, $(r_{j2} : GL_{j2} \Rightarrow_{a_{j2}} GR_{j2}) \in \mathcal{R}_0$. There exists graphs G_1 and G_2, specialization s_1 from $GL_{j1}{+}G_1$ to G and specialization s_2 from $GL_{j2}+G_2$ to G, such that $(G'_1, r_1, \_)$ is the pushout of $(r_{j1}+\mathrm{id}_{G_1}, s_1)$ and $(G'_2, r_2, \_)$ is the pushout of $(r_{j2}+\mathrm{id}_{G_2}, s_2)$.

Now, by the second requirement to the level 0-rules, $j1 = j2$. Then, by the second part of the first requirement to the level 0-rules, we from $s_1(\mathsf{in}_1(a_{j1})) = s_2(\mathsf{in}_1(a_{j1}))$ can conclude that $s_1(\mathsf{in}_1(n)) = s_2(\mathsf{in}_1(n))$ for all n in GL_{j1} (since s_1 and s_2 respect all active and passive nodes). Finally, apply lemma 4.1.41 to find isomorphism h from G'_1 to G'_2 such that $r_1 \star h = r_2$. □

We can now formulate a "diamond lemma":

Lemma 4.4.3 Suppose $1 \vdash (j1)\ r_1 : G \Rightarrow_{a_1} G'_1$ and $1 \vdash (j2)\ r_2 : G \Rightarrow_{a_2} G'_2$. Then one of two holds:

1. $a_1 = a_2$, $j1 = j2$, $r_1 = r_2$ and $G'_1 = G'_2$ (modulo isomorphism).
2. $a_1 \neq a_2$. Then there exist r'_1, r'_2 and G' such that

   $$1 \vdash (j2)\ r'_1 : G'_1 \Rightarrow_{r_1(a_2)} G', 1 \vdash (j1)\ r'_2 : G'_2 \Rightarrow_{r_2(a_1)} G'$$

   and such that $r_1 \star r'_1 = r_2 \star r'_2$.

□

Proof: If $a_1 = a_2$ the claim follows from lemma 4.4.2, so assume $a_1 \neq a_2$. Let

$$(r_{j1} : GL_{j1} \Rightarrow_{a_{j1}} GR_{j1}) \in \mathcal{R}_0, (r_{j2} : GL_{j2} \Rightarrow_{a_{j2}} GR_{j2}) \in \mathcal{R}_0.$$

There exists graphs G_1, G_2 and specializations s_1 from $G_{j1}+G_1$ to G, s_2 from $G_{j2}+G_2$ to G such that $(G'_1, r_1, \_)$ is the pushout of $(r_{j1}+\mathrm{id}_{G_1}, s_1)$, $(G'_2, r_2, \_)$ is the pushout of $(r_{j2}+\mathrm{id}_{G_2}, s_2)$ and $s_1(\mathsf{in}_1(a_{j1})) = a_1$, $s_2(\mathsf{in}_1(a_{j2})) = a_2$. Now define m, a morphism from $GL_{j1}+GL_{j2}$ to G, by stipulating m to coincide with s_1 on GL_{j1} and to coincide with s_2 on GL_{j2}. We want to apply lemma 4.3.12; to this end we must show

- that m respects active and passive nodes, which is trivial;
- that m satisfies `INC`, which is trivial;
- that m satisfies SPEC, which is trivial;

- that if $m(a) = m(a')$ then a and a' both are partial applications. If e.g. a and a' both come from GL_{j1}, then both are partial applications since s_1 is a specialization. So assume that a comes from GL_{j1} and a' comes from GL_{j2}. For the sake of a contradiction we assume that a is not a partial application. Then, due to the first requirement to level 0 rules, $a = a_{j1}$ and we have $a_1 = m(a')$. Now a' is either a genuine partial application or equal a_{j2}; in the former case we find that also $m(a') = a_1$ is a genuine partial application (contradiction) and in the latter case we find $a_1 = a_2$ (contradiction).

So there exists G_3 and specialization s from $GL_{j1}+GL_{j2}+G_3$ to G, such that s and s_1 coincide on GL_{j1}, and such that s and s_2 coincide on GL_{j2}.

The situation is as depicted in figure 4.13, where $r'_{j1} = \mathrm{id}_{G_3}+r_{j1}+\mathrm{id}_{GL_{j2}}$, $r''_{j1} = \mathrm{id}_{G_3}+\mathrm{id}_{GR_{j1}}+r_{j2}$, $r'_{j2} = \mathrm{id}_{G_3}+\mathrm{id}_{GL_{j1}}+r_{j2}$ and $r''_{j2} = \mathrm{id}_{G_3}+r_{j1}+\mathrm{id}_{GR_{j2}}$. Here we have (re)defined (G'_1, r_1, s'_1) as the pushout of (r'_{j1}, s); that this coincides (modulo isomorphism) with the "old" definition of G'_1 and r_1 is a consequence of lemma 4.4.2 – likewise for G'_2 and r_2. Then we defined (G''_1, r'_1, s''_1) as the pushout of (r''_{j1}, s'_1) and likewise (G''_2, r'_2, s''_2) as the pushout of (r''_{j2}, s'_2).

We have $r'_{j1}\star r''_{j1} = r'_{j2}\star r''_{j2}(= \mathrm{id}_{G_3}+r_{j1}+r_{j2})$. According to fact 4.1.23, 4, the pushout of $(r'_{j1}\star r''_{j1}, s)$ is $(G''_1, r_1\star r'_1, s''_1)$ and the pushout of $(r'_{j2}\star r''_{j2}, s)$ is $(G''_2, r_2\star r'_2, s''_2)$. Hence $G''_1 = G''_2$ (which we can take to be G'), and $r_1\star r'_1 = r_2\star r'_2$. What is left (in order to show $1 \vdash (j2)\ r'_1 : G'_1 \Rightarrow_{r_1(a_2)} G'$) is to show that $s'_1(\mathsf{in\_}(a_{j2})) = r_1(a_2)$. But this is an immediate consequence of $s\star r_1 = r'_{j1}\star s'_1$. □

Then we can show that the transition system is confluent:

Theorem 4.4.4 Suppose $1 \vdash^* r_1 : G \Rightarrow^{c_1}_{Nn_1} G_1$ and $1 \vdash^* r_2 : G \Rightarrow^{c_2}_{Nn_2} G_2$. Then there exists G', r'_1, r'_2, n'_1, n'_2, c'_1 and c'_2 such that

$$1 \vdash^* r'_1 : G_1 \Rightarrow^{c'_1}_{Nn'_1} G', 1 \vdash^* r'_2 : G_2 \Rightarrow^{c'_2}_{Nn'_2} G'$$

Moreover, $r_1\star r'_1 = r_2\star r'_2$ and $c_1 + c'_1 = c_2 + c'_2$. Finally, suppose $c_1 = n_1$ and $c_2 = n_2$ (i.e. r_1 and r_2 are normal order reductions) – then $c'_1 = n'_1$, $c'_2 = n'_2$. □

Proof: First notice that we have the following property (where lemma 4.4.3 caters for the case $c_1 = 1$ and $c_2 = 1$, as then either $r'_1 = r'_2 = \mathrm{id}\_$, $c'_1 = c'_2 = 0$ or $c'_1 = c'_2 = 1$):

$$\begin{tikzcd}
\begin{array}{c} G_3+ \\ GL_{j1}+GL_{j2} \end{array} \arrow[r, "r'_{j1}"] \arrow[dd, "s"'] & \begin{array}{c} G_3+ \\ GR_{j1}+GL_{j2} \end{array} \arrow[r, "r''_{j1}"] \arrow[d, "s'_1"] & \begin{array}{c} G_3+ \\ GR_{j1}+GR_{j2} \end{array} \arrow[d, "s''_1"] \\
& G'_1 \arrow[r, "r'_1"] & G''_1 \\
G \arrow[ur, "r_1"] \arrow[dr, "r_2"'] & & \\
& G'_2 \arrow[r, "r'_2"'] & G''_2 \\
\begin{array}{c} GL_{j1}+GL_{j2} \\ +G_3 \end{array} \arrow[r, "r'_{j2}"'] \arrow[uu, "s"] & \begin{array}{c} GL_{j1}+GR_{j2} \\ +G_3 \end{array} \arrow[r, "r''_{j2}"'] \arrow[u, "s'_2"'] & \begin{array}{c} GR_{j1}+GR_{j2} \\ +G_3 \end{array} \arrow[u, "s''_2"']
\end{tikzcd}$$

Figure 4.13: A proof of the diamond property.

> If $1 \vdash^* r_1 : G \Rightarrow^{c_1}_{Nn_1} G_1$ and $1 \vdash^* r_2 : G \Rightarrow^{c_2}_{Nn_2} G_2$, with c_1 either 0 or 1 and c_2 either 0 or 1, then there exists G', r'_1, r'_2, and $c'_1, c'_2, n'_1, n'_2 \in \{0, 1\}$ such that $1 \vdash^* r'_1 : G_1 \Rightarrow^{c'_1}_{Nn'_1} G'$ and $1 \vdash^* r'_2 : G_2 \Rightarrow^{c'_2}_{Nn'_2} G'$. Moreover, $r_1 \star r'_1 = r_2 \star r'_2$ and $c_1 + c'_1 = c_2 + c'_2$. Finally, if $c_1 = n_1$ and $c_2 = n_2$ then $c'_1 = n'_1$ and $c'_2 = n'_2$.

Then the theorem easily follows by "completing the lattice", as suggested in figure 4.14. □

4.4.1 Normal forms

When defining a transition system, a distinguished subset of the configurations are termed "normal forms". At first glance, it seems natural to say that G is in normal form iff G contains no redices, since this amounts to saying that there exists no G' such that $1 \vdash (j)\ r : G \Rightarrow_a G'$. However, in the absence of "garbage collection" this is too restricted: some redices which are no longer needed may prevent a graph from being in normal form.

$$\begin{tikzcd}[column sep=small]
 & & G \arrow[dl, "r_1"'] \arrow[dr, "r_2"] & & \\
 & \bullet \arrow[dl] \arrow[dr] & & \bullet \arrow[dl] \arrow[dr] & \\
G_1 \arrow[dr] & & \bullet \arrow[dl] \arrow[dr] & & G_2 \arrow[dl] \\
 & \bullet \arrow[dr, "r_1'"'] & & \bullet \arrow[dl, "r_2'"] & \\
 & & G' & &
\end{tikzcd}$$

Figure 4.14: The confluence property.

Therefore we rather settle on (where G is equipped with demand function $\mathcal{D}$):

Definition 4.4.5 A graph G is in *normal form* if it contains no redex a with $\mathcal{D}(a) \geq 1$. □

Another condition turns out to be of interest:

Definition 4.4.6 A graph G is said to be in *well-typed normal form* if all nodes n with $\mathcal{D}(n) = 2$ are passive. □

For the connection between normal forms and well-typed normal forms, we have

Fact 4.4.7 Suppose G has result node $n0$, and is in well-typed normal form. Then also G is in normal form. □

PROOF: We have that $\mathcal{D} = \mathit{Clo}[G](\mathsf{D}_{n0})$. It will be sufficient to show that a is a partial application if $\mathcal{D}(a) \geq 1$, i.e. if $\vdash \mathcal{D}(a) \geq d$ with $d \in \{1, 2\}$ using the inference system in figure 4.9. Below, this will be done by induction in the proof tree.

If $\vdash \mathcal{D}(a) \geq d$ because $\mathsf{D}_{n0}(a) = d$, then $d = 2$ (and $a = n0$). As G is in well-typed normal form, a is passive – but this is a contradiction.

If $\vdash \mathcal{D}(a) \geq 1$ because $\vdash \mathcal{D}(a) \geq 2$, or if $\vdash \mathcal{D}(a) \geq 2$ because $a = \mathcal{S}(p, i)$, $\vdash \mathcal{D}(p) \geq 2$ then again well-typedness tells us that a is passive, again yielding a contradiction.

If $\vdash \mathcal{D}(a) \geq 1$ because $a = \mathcal{S}(a', 1)$, $\vdash \mathcal{D}(a') \geq 1$ then by the induction hypothesis a' is a partial application – hence also a is a partial application.

If $\vdash \mathcal{D}(a) \geq 1$ because $Ndarg(a', a)$, $\vdash \mathcal{D}(a') \geq 1$ then by induction a' is a partial application - but this yields a contradiction.

If $\vdash \mathcal{D}(a) \geq 1$ because a is passive or a genuine partial application, then clearly a is a partial application! $\square$

For the opposite direction, suppose G has result node $n0$ and is in normal form but not in well-typed normal form. This could for instance happen if $n0$ is a cons-cell with partial applications as children. Then it seems fair to say that G is "ill-typed" in the sense that "the result node cannot be assigned a first-order type"[9]. Recall from the beginning of this chapter that we do not want to formulate any type system.

Definition 4.4.8 Let G be *singlelabeled* and in well-typed normal form, and let n be a node in G with $\mathcal{D}(n) = 2$. Then we can define $\mathrm{Val}_G(n)$, "the value of n", as follows: let $l = \mathcal{L}(n)$ (exists as n is passive), let $a = Ar(l)$, and let $\{n_i | i \in \{1 \ldots a\}\}$ be defined by $n_i = \mathcal{S}(n, i)$. As $\mathcal{D}(n_i) = 2$, it makes sense to define

$$\mathrm{Val}_G(n) = l(\mathrm{Val}_G(n_1), \ldots, \mathrm{Val}_G(n_a))$$

($\mathrm{Val}_G(n)$ may be an infinite term, if G is cyclic). $\square$

As already mentioned, even if G is in normal form there may exist G' such that $1 \vdash (j)\ r : G \Rightarrow_a G'$. However, for our purposes the content of the following lemma will be sufficient:

Lemma 4.4.9 Suppose G is in normal form and $1 \vdash^* r : G \Rightarrow^c_{Nn} G'$. Then G' is in normal form, and $n = 0$. G' is in well-typed normal form if and only if G is, in which case also – provided G and hence also G' is singlelabeled – $\mathrm{Val}_{G'}(r(n)) = \mathrm{Val}_G(n)$ for n such that $\mathcal{D}(n) = 2$. $\square$

PROOF: It will be enough to show that if $1 \vdash (j)\ r : G \Rightarrow_a G'$ and G is in normal form then

1. $\mathcal{D}(a) = 0$.
2. G' is in normal form.
3. G' is in well-typed normal form iff G is.
4. If G is in well-typed normal form and singlelabeled, then $\mathrm{Val}_{G'}(r(n)) = \mathrm{Val}_G(n)$ for n with $\mathcal{D}(n) = 2$.

[9]Notice that we are *not* saying that it should be possible to assing an *arbitrary* node a first-order type.

1 is straightforward (as G contains no redex a with $\mathcal{D}(a) \geq 1$). Then 2 follows by lemma 4.3.13. 4 is immediate, as r respects passive nodes (but this does not show the "if"-part of 3, since in principle G' might contain non-passive nodes n with $\mathcal{D}'(n) = 2$). To show 3, proceed as follows:

"if" Suppose G is in well-typed normal form. It will be sufficient to prove the following: if $\vdash \mathcal{D}'(n') \geq 2$ using the inference system in figure 4.9 (where $\mathsf{D}'(n') = \max\{\mathcal{D}(n) | r(n) = n'\}$), then there exists n in G with $\mathcal{D}(n) = 2$ such that $r(n) = n'$ (and hence n' is passive, as n is passive). This will be done by induction in the proof tree, with only two non-trivial cases:

- Suppose $\vdash \mathcal{D}'(n') \geq 2$ because $n' = \mathcal{S}'(p', i)$, $\vdash \mathcal{D}'(p') \geq 2$. By induction, there exists p in G with $\mathcal{D}(p) = 2$ such that $r(p) = p'$. Let $n = \mathcal{S}(p, i)$, then $\mathcal{D}(n) = 2$ and $r(n) = n'$.
- Suppose $\vdash \mathcal{D}'(n') \geq 2$ because $\mathsf{D}'(n') = 2$. But this just means that there exists n in G such that $\mathcal{D}(n) = 2$, $r(n) = n'$.

"only if" Suppose G is *not* in well-typed normal form. Then there exists a node n with $\mathcal{D}(n) = 2$ which is *not* passive. As G is in normal form, n is not a redex (and in particular not equal a). But then, as r respects all nodes but a, $r(n)$ is not passive – yet $\mathcal{D}'(r(n)) = 2$, showing that G' is not in well-typed normal form.

□

The following lemma states that one is not able to arrive at a normal form by doing a non-normal order step:

Lemma 4.4.10 Suppose $1 \vdash (j)\ r : G \Rightarrow_a G'$, with G' in normal form and with $\mathcal{D}(a) = 0$. Then also G is in normal form. □

Proof: Suppose G contains a redex a_1 with $\mathcal{D}(a_1) \geq 1$. Then, as r respects all nodes but a and $a \neq a_1$, we conclude that $r(a_1)$ is a redex in G' with $\mathcal{D}'(r(a_1)) \geq 1$ – i.e. G' is not in normal form. □

We will expect "normal forms to be unique". In some sense, this is the content of the following theorem:

Theorem 4.4.11 Let G be a singlelabeled graph with result node $n0$. Suppose $1 \vdash^* r_1 : G \Rightarrow_{\bar{N}_-} G_1$ and $1 \vdash^* r_2 : G \Rightarrow_{\bar{N}_-} G_2$, with G_1 and G_2 in well-typed normal form. Then $\mathrm{Val}_{G_1}(r_1(n0)) = \mathrm{Val}_{G_2}(r_2(n0))$. □

PROOF: By theorem 4.4.4, there exist G', r'_1 and r'_2 such that $1 \vdash^* r'_1 : G_1 \Rightarrow_{\bar{N}\_} G'$, $1 \vdash^* r'_2 : G_2 \Rightarrow_{\bar{N}\_} G'$, and such that $r_1 \star r'_1 = r_2 \star r'_2$. By fact 4.3.14, $r_1(n0)$ ($r_2(n0)$) is a result node of G_1 (G_2), and then by fact 4.4.7 G_1 and G_2 are in normal form. By lemma 4.4.9, G' is in well-typed normal form and $\mathrm{Val}_{G'}(r'_1(r_1(n0))) = \mathrm{Val}_{G_1}(r_1(n0))$, $\mathrm{Val}_{G'}(r'_2(r_2(n0))) = \mathrm{Val}_{G_2}(r_2(n0))$. But this shows that $\mathrm{Val}_{G_1}(r_1(n0)) = \mathrm{Val}_{G_2}(r_2(n0))$. □
Next we show that "all normal order reductions have equal length":

Lemma 4.4.12 Suppose $1 \vdash^* r_1 : G \Rightarrow^n_{Nn} G_1$ and $1 \vdash^* r_2 : G \Rightarrow^n_{Nn} G_2$. Then G_1 is in normal form iff G_2 is in normal form, in which case $G_1 = G_2$. □

PROOF: By theorem 4.4.4, we find that there exists G', r'_1, r'_2 and n' such that $1 \vdash^* r'_1 : G_1 \Rightarrow^{n'}_{Nn'} G'$, $1 \vdash^* r'_2 : G_2 \Rightarrow^{n'}_{Nn'} G'$. Suppose G_1 is in normal form. Then it must hold that $n' = 0$, hence $G_2 = G_1$. □
The following lemma says that instead of first doing a non-normal order step and then a normal order step, one can do the normal order step first:

Lemma 4.4.13 Let G be singlelabeled. Suppose $1 \vdash (j)\ r_1 : G \Rightarrow_a G_1$ and $1 \vdash (j_1)\ r_2 : G_1 \Rightarrow_{a'_1} G_2$, with $\mathcal{D}(a) = 0$ and $\mathcal{D}_1(a'_1) \geq 1$ (where $\mathcal{D}$ ($\mathcal{D}_1$) is the demand function of G (G_1)). Then there exist r'_1, r'_2, G'_1 and a' with $r_1(a') = a'_1$ and $\mathcal{D}(a') \geq 1$ such that

$$1 \vdash (j_1)\ r'_1 : G \Rightarrow_{a'} G'_1, 1 \vdash (j)\ r'_2 : G'_1 \Rightarrow_{r'_1(a)} G_2$$

and $r_1 \star r_2 = r'_1 \star r'_2$. □

PROOF: Lemma 4.3.13 tells us that there exists a' in G with $r_1(a') = a'_1$ such that $\mathcal{D}(a') \geq 1$ and a' is a redex. Hence (as G is singlelabeled) there exists r'_1 and G'_1 such that $1 \vdash (j'_1)\ r'_1 : G \Rightarrow_{a'} G'_1$ (for some j'_1). $a \neq a'$, so lemma 4.4.3 applied to r_1 and r'_1 says that there exists r'_2,r''_2 and G'_2 such that

$$1 \vdash (j)\ r'_2 : G'_1 \Rightarrow_{r'_1(a)} G'_2, 1 \vdash (j'_1)\ r''_2 : G_1 \Rightarrow_{r_1(a')} G'_2$$

and $r'_1 \star r'_2 = r_1 \star r''_2$. Now, by lemma 4.4.2 applied to r_2 and r''_2 exploiting that $r_1(a') = a'_1$, we have $r_2 = r''_2$, $G'_2 = G_2$ and $j_1 = j'_1$. Hence the claim.
□
We now can show that "normal order reduction is optimal":

Theorem 4.4.14 Let G be singlelabeled. Suppose $1 \vdash^* r : G \Rightarrow^c_{Nn} G'$, with G' in normal form. Then there exists G'' in normal form, reduction r' and c' with $n \leq c' \leq c$, such that $1 \vdash^* r' : G \Rightarrow^{c'}_{Nc'} G''$. Moreover, G'' is in well-typed normal form iff G' is. □

PROOF: By repeated applications of lemma 4.4.13, we find that there exist r', r'', G'', $c' \geq n$ and c'' such that $1 \vdash^* r' : G \Rightarrow^{c'}_{Nc'} G''$, $1 \vdash^* r'' : G'' \Rightarrow^{c''}_{N0} G'$, $r' {\star} r'' = r$ and $c' + c'' = c$.

By repeated application of lemma 4.4.10, we see that G'' is in normal form. The last claim of the theorem follows from lemma 4.4.9. □
In a similar vein, we can show that "if there exists a looping evaluation with arbitrary many normal order steps, then also normal order reduction will loop" First we make the following

Definition 4.4.15 Given G (singlelabeled). We say that G *loops at level 1 by a normal order strategy*, if for all n and G' such that

$$1 \vdash^* \_ : G \Rightarrow^{n}_{Nn} G'$$

G' is *not* in normal form. □

Theorem 4.4.16 Let singlelabeled graph G be given. Suppose for all n, there exists $n_1 \geq n$, r_1 and G_1 such that $1 \vdash^* r_1 : G \Rightarrow^{-}_{Nn_1} G_1$. Then G loops at level 1 by a normal order strategy. □

PROOF: Let $1 \vdash^* \_ : G \Rightarrow^{n}_{Nn} G'$; we want to show that G' is not in normal form. By assumption, there exists r_1, G_1 and $n_1 \geq n+1$ such that $1 \vdash^* r_1 : G \Rightarrow^{-}_{Nn_1} G_1$. By repeated application of lemma 4.4.13, we find G'_1 and $n'_1 \geq n_1$ such that $1 \vdash^* \_ : G \Rightarrow^{n'_1}_{Nn'_1} G'_1$. Hence, as $n'_1 \geq n+1$, we find G'' and $c \geq 1$ such that

$$1 \vdash^* \_ : G \Rightarrow^{n}_{Nn} G'' \quad \text{and} \quad 1 \vdash^* \_ : G'' \Rightarrow^{c}_{Nc} G'_1$$

the right hand side of which shows that G'' is not in normal form. Now apply lemma 4.4.12 to the left hand side. □

4.5 Transitions at level i

Now we embark on defining the level i transitions, $i > 1$. We shall write

$$i \vdash r : G \Rightarrow_{Nn} G'$$

to denote that r, a reduction from G to G', is a "single step" at level i which "represents n normal order steps at level 1". We shall write

$$i \vdash^* r : G \Rightarrow^{c}_{Nn} G'$$

to denote that r, a reduction from G to G', consists of c steps at level i and "represents n normal order steps at level 1" (it should be noticed that it may happen that $n > c$). Finally, we shall write (for $i \geq 1$)

$$(r : G \Rightarrow^{c}_{Nn} G') \in \mathcal{R}_i$$

to denote that r, a reduction from G to G', is a level i rule.

- The following inference system determines when it holds that $i \vdash^* r : G \Rightarrow^{c}_{Nn} G'$ $(i > 1)$:

$$\frac{i \vdash r : G \Rightarrow_{Nn} G'}{i \vdash^* r : G \Rightarrow^{1}_{Nn} G'}$$

$$i \vdash^* \mathrm{id}_G : G \Rightarrow^{0}_{N0} G$$

$$\frac{i \vdash^* r_1 : G_1 \Rightarrow^{c_1}_{Nn_1} G_2, i \vdash^* r_2 : G_2 \Rightarrow^{c_2}_{Nn_2} G_3}{i \vdash^* r_1 \star r_2 : G_1 \Rightarrow^{(c_1+c_2)}_{N(n_1+n_2)} G_3}$$

- Concerning $\mathcal{R}_i$, the set of level i rules ($i \geq 1$), we must demand:

  1. if $(r : G \Rightarrow^{c}_{Nn} G') \in \mathcal{R}_i$, then $i \vdash^* r : G \Rightarrow^{c}_{Nn} G'$;
  2. $\mathcal{R}_i$ is a finite set for all $i \geq 1$.

  The pragmatics behind 2 is that it will not be possible (in finite time) to generate infinitely many rules.

- $i \vdash r : G \Rightarrow_{Nn} G'$ $(i > 1)$ will hold provided (for some G_1,G'_1,G_2,s and r_1) $(G', r, \_)$ is the pushout of $(r_1 + \mathrm{id}_{G_2}, s)$ where s is a specialization from $G_1 + G_2$ to G, where r_1 is a reduction from G_1 to G'_1, and where *either*

  1. $(r_1 : G_1 \Rightarrow_{a_1} G'_1) \in \mathcal{R}_0$ – then $n = 1$ if $\mathcal{D}(s(\mathsf{in}_1(a_1))) \geq 1$, otherwise $n = 0$ – *or*
  2. $(r_1 : G_1 \Rightarrow^{c_1}_{Nn_1} G'_1) \in \mathcal{R}_{i'}$ for some $i' < i$ – then $n = n_1$.

  This captures the intuition that "rules at lower levels can be exploited". As discussed in chapter 2, in concrete multilevel systems there are often restrictions on *which* lower level rules can be used.

It is convenient to note that the above definition of $i \vdash r : G \Rightarrow_{Nn} G'$ is also applicable to the case $i = 1$. Then we find, with $n = 1$ if $\mathcal{D}(a) \geq 1$ and $n = 0$ otherwise, that

$$1 \vdash r : G \Rightarrow_{Nn} G' \Leftrightarrow \exists j : 1 \vdash (j)\ r : G \Rightarrow_a G'$$

Now also the above definition of $i \vdash^* r : G \Rightarrow^c_{Nn} G'$ can be extended to the case $i = 1$.

Properties of level i transitions

The following lemmas express that "if r is a reduction more general than r_1, and r is a level i transition, then so is r_1". They are all easy consequences of the algebraic laws stated in section 4.1.

Lemma 4.5.1 Suppose $i \vdash r : G \Rightarrow_{Nn} G'$, $i \geq 1$. Suppose s is a specialization from G to G_1. Let $(G'_1, r_1, \_)$ be the pushout of (r, s). Then also $i \vdash r_1 : G_1 \Rightarrow_{Nn_1} G'_1$, with $n_1 \geq n$. □

PROOF: We have that $(G', r, \_)$ is the pushout of $(r_2{+}\mathrm{id}_{G_3}, s_2)$ where s_2 is a specialization from $G_2{+}G_3$ to G, and where *either*

1. $(r_2 : G_2 \Rightarrow_{a_2} G'_2) \in \mathcal{R}_0$, $n = 1$ if $\mathcal{D}(s_2(\mathsf{in}_1(a_2))) \geq 1$, $n = 0$ otherwise *or*

2. $(r_2 : G_2 \Rightarrow^{c_2}_{Nn_2} G'_2) \in \mathcal{R}_{i'}$ for some $i' < i$, $n = n_2$.

In both cases, we then exploit that – by fact 4.1.23 (4.3) – $(G'_1, r_1, \_)$ is the pushout of $(r_2{+}\mathrm{id}_{G_3}, s_2{\star}s)$. Hence $i \vdash r_1 : G_1 \Rightarrow_{Nn_1} G'_1$ for some n_1. In case 1, we have $n_1 = 1$ if $\mathcal{D}_1(s(s_2(\mathsf{in}_1(a_2)))) \geq 1$, $n_1 = 0$ otherwise – as s satisfies `INC` this shows that $n_1 \geq n$. In case 2, we clearly have $n_1 = n_2 = n$. □

Lemma 4.5.2 Suppose $i \vdash^* r : G \Rightarrow^c_{Nn} G'$, $i \geq 1$. Suppose s is a specialization from G to G_1. Let $(G'_1, r_1, \_)$ be the pushout of (r, s). Then also $i \vdash^* r_1 : G_1 \Rightarrow^c_{Nn_1} G'_1$, with $n_1 \geq n$. □

PROOF: Induction in the proof tree for $i \vdash^* r : G \Rightarrow^c_{Nn} G'$. Three cases:

1. We have $c = 1$ and $i \vdash r : G \Rightarrow_{Nn} G'$. By lemma 4.5.1 there exists $n_1 \geq n$ such that $i \vdash r_1 : G_1 \Rightarrow_{Nn_1} G'_1$, and hence also
   $i \vdash^* r_1 : G_1 \Rightarrow^1_{Nn_1} G'_1$.

2. We have $G' = G$, $r = \mathrm{id}_G$ and $c = n = 0$. By fact 4.1.23 (4.2) we have $G'_1 = G_1$, $r_1 = \mathrm{id}_{G_1}$. Hence $i \vdash^* r_1 : G_1 \Rightarrow^0_{N0} G'_1$, as desired.

3. We have $i \vdash^* r'' : G \Rightarrow^{c''}_{Nn''} G''$ and $i \vdash^* r' : G'' \Rightarrow^{c'}_{Nn'} G'$, $r = r''\star r'$, $c = c''+c'$ and $n = n''+n'$. Let (G''_1, r''_1, s''_1) be the pushout of (r'', s). Then, by fact 4.1.23 (4.4), the pushout of (r', s''_1) is $(G'_1, r'_1, \_)$ where $r_1 = r''_1\star r'_1$. By induction, there exists $n''_1 \geq n''$, $n'_1 \geq n'$ such that $i \vdash^* r''_1 : G_1 \Rightarrow^{c''}_{Nn''_1} G''_1$, $i \vdash^* r'_1 : G''_1 \Rightarrow^{c'}_{Nn'_1} G'_1$. Hence we have $i \vdash^* r_1 : G_1 \Rightarrow^{c}_{Nn_1} G'_1$, with $n_1 = n''_1 + n'_1 \geq n'' + n' = n$.

□

Lemma 4.5.3 Suppose $i \vdash r : G \Rightarrow_{Nn} G'$, $i \geq 1$. Then for all G_1, also $i \vdash r+\mathrm{id}_{G_1} : G+G_1 \Rightarrow_{Nn} G'+G_1$ □

PROOF: We have that $(G', r, \_)$ is the pushout of $(r_2+\mathrm{id}_{G_3}, s_2)$ where s_2 is a specialization from G_2+G_3 to G, and where *either*

1. $(r_2 : G_2 \Rightarrow_{a_2} G'_2) \in \mathcal{R}_0$, $n = 1$ if $\mathcal{D}(s_2(\mathsf{in}_1(a_2))) \geq 1$, $n = 0$ otherwise *or*

2. $(r_2 : G_2 \Rightarrow^{c_2}_{Nn_2} G'_2) \in \mathcal{R}_{i'}$ for some $i' < i$, $n = n_2$.

Now $s_2+\mathrm{id}_{G_1}$ is a specialization from $G_2+G_3+G_1$ to $G+G_1$. By fact 4.1.24, the pushout of $(r_2+\mathrm{id}_{G_3}+\mathrm{id}_{G_1}, s_2+\mathrm{id}_{G_1})$ is $(G'+G_1, r+\mathrm{id}_{G_1}, \_)$. This shows that $i \vdash r+\mathrm{id}_{G_1} : G+G_1 \Rightarrow_{Nn} G'+G_1$. □

Lemma 4.5.4 Suppose $i \vdash^* r : G \Rightarrow^c_{Nn} G'$, $i \geq 1$. Then for all G_1, also $i \vdash^* r+\mathrm{id}_{G_1} : G+G_1 \Rightarrow^c_{Nn} G'+G_1$. □

PROOF: Induction in the proof tree for $i \vdash^* r : G \Rightarrow^c_{Nn} G'$. Three cases:

1. We have $c = 1$ and $i \vdash r : G \Rightarrow_{Nn} G'$. By lemma 4.5.3,
   $i \vdash r+\mathrm{id}_{G_1} : G+G_1 \Rightarrow_{Nn} G'+G_1$, and hence also
   $i \vdash^* r+\mathrm{id}_{G_1} : G+G_1 \Rightarrow^1_{Nn} G'+G_1$.

2. We have $G' = G$, $r = \mathrm{id}_G$ and $c = n = 0$. As $\mathrm{id}_G+\mathrm{id}_{G_1} = \mathrm{id}_{G+G_1}$, we clearly have $i \vdash^* r+\mathrm{id}_{G_1} : G+G_1 \Rightarrow^0_{N0} G'+G_1$.

3. We have $i \vdash^* r'' : G \Rightarrow^{c''}_{Nn''} G''$ and $i \vdash^* r' : G'' \Rightarrow^{c'}_{Nn'} G'$, $r = r''\star r'$, $c = c'' + c'$ and $n = n'' + n'$. By induction, we have

   $i \vdash^* r''+\mathrm{id}_{G_1} : G+G_1 \Rightarrow^{c''}_{Nn''} G''+G_1$ and
   $i \vdash^* r'+\mathrm{id}_{G_1} : G''+G_1 \Rightarrow^{c'}_{Nn'} G'+G_1$.

By fact 4.1.15 we – as desired – get $i \vdash^* r{+}\mathrm{id}_{G_1} : G{+}G_1 \Rightarrow^c_{Nn} G'{+}G_1$.

□

Corollary 4.5.5 Suppose $i \vdash^* r_1 : G_1 \Rightarrow^{c_1}_{Nn_1} G'_1$ and $i \vdash^* r_2 : G_2 \Rightarrow^{c_2}_{Nn_2} G'_2$, $i \geq 1$. Then

$$i \vdash^* r_1{+}r_2 : G_1{+}G_2 \Rightarrow^{(c_1+c_2)}_{N(n_1+n_2)} G'_1{+}G'_2$$

□

Proof: By lemma 4.5.4, we have

$$i \vdash^* r_1{+}\mathrm{id}_{G_2} : G_1{+}G_2 \Rightarrow^{c_1}_{Nn_1} G'_1{+}G_2,\ i \vdash^* \mathrm{id}_{G'_1}{+}r_2 : G'_1{+}G_2 \Rightarrow^{c_2}_{Nn_2} G'_1{+}G'_2$$

By fact 4.1.15, we have $(r_1{+}\mathrm{id}_{G_2})\star(\mathrm{id}_{G'_1}{+}r_2) = r_1{+}r_2$ – hence the claim. □

4.6 Correctness and speedup bounds

In this section we will investigate the relationship between the various levels present in a multilevel system. In particular, we will

1. give bounds for the speedup one can expect to gain when working at level i instead of level 1;
2. give conditions for a multilevel system to be "correct", in the sense that "working at level i gives the same result as working at level 1" – the nontrivial point is to ensure that working at level i does not increase the risk of nontermination.

First some notation:

- for $i \geq 1$, define $\mathcal{C}_i$ as the maximum of the c's such that there exists a rule $(\_ : \_ \Rightarrow^c_{N\_}\ \_) \in \mathcal{R}_i$ – however, if this maximum is 0 we stipulate $\mathcal{C}_i = 1$. Since we required each $\mathcal{R}_i$ to be finite, $\mathcal{C}_i < \infty$. As rules represent shortcuts in the computation process, the intuition is that $\mathcal{C}_i$ is "the maximal shortcut represented by a level i rule".
- for $i \geq 1$, define $\mathcal{T}_i$ as follows: Let $\{(r_j, G_j, G'_j, c_j, n_j) | j \in J\}$ be such that $(r : G \Rightarrow^c_{Nn} G') \in \mathcal{R}_i$ iff there exists $j \in J$ with $r = r_j$, $G = G_j$, $G' = G'_j$, $c = c_j$ and $n = n_j$. Then we stipulate

  $$\mathcal{T}_i = \sum_{j \in J} c_j + 1$$

(the "+1" is added for technical reasons and will often be dispensed with in examples.) One should think of $\mathcal{T}_i$ as denoting "the total cost of deriving the level i rules", as intuitively the cost of deriving a rule is proportional to the shortcut it represents.

- for $i \geq 1$, we define

  $$\mathcal{T}\mathcal{T}_i = \sum_{j=1}^{i} \mathcal{T}_j$$

  to be interpreted as the total cost of deriving the rules at level $\leq i$.

Next we are – hardly surprising! – able to show that "level i can simulate level $i+1$":

Lemma 4.6.1 Suppose $i+1 \vdash^* r : G \Rightarrow^{c}_{Nn} G'$, $i \geq 1$. Then there exists $c' \leq \mathcal{C}_i \cdot c$, $n' \geq n$ such that $i \vdash^* r : G \Rightarrow^{c'}_{Nn'} G'$. □

PROOF: We will use induction in the proof tree for $i+1 \vdash^* r : G \Rightarrow^{c}_{Nn} G'$. Three cases:

- A rule at level $i' < i+1$ has been exploited, then $c = 1$. Two cases:
  - $i' < i$. Then we also have $i \vdash^* r : G \Rightarrow^{c}_{Nn} G'$, and as $\mathcal{C}_i \geq 1$ we have $c \leq \mathcal{C}_i c$.
  - $i' = i$ (so $i' \neq 0$). Then there exists $(r_1 : G_1 \Rightarrow^{c_1}_{Nn_1} G'_1) \in \mathcal{R}_i$, G_2 and specialization s from G_1+G_2 to G, such that $(G', r, \_)$ is the pushout of $(r_1+\mathrm{id}_{G_2}, s)$. Moreover $n_1 = n$.

    By assumption, we have $i \vdash^* r_1 : G_1 \Rightarrow^{c_1}_{Nn} G'_1$. By lemma 4.5.4, we have

    $$i \vdash^* r_1+\mathrm{id}_{G_2} : G_1+G_2 \Rightarrow^{c_1}_{Nn} G'_1+G_2$$

    and by lemma 4.5.2 we then find that there exists $n' \geq n$ such that

    $$i \vdash^* r : G \Rightarrow^{c_1}_{Nn'} G'$$

    As $c_1 \leq \mathcal{C}_i$ by definition, we finally obtain $c_1 \leq \mathcal{C}_i c$ as desired.

- We have $G' = G$, $r = \mathrm{id}_G$ and $c = n = 0$. But then clearly $i \vdash^* r : G \Rightarrow^{0}_{N0} G'$ – and $0 \leq \mathcal{C}_i 0$, $0 \geq 0$.

- We have $i+1 \vdash^* r_1 : G \Rightarrow^{c_1}_{Nn_1} G''$, $i+1 \vdash^* r_2 : G'' \Rightarrow^{c_2}_{Nn_2} G'$ with $r = r_1 {\star} r_2$, $c = c_1 + c_2$ and $n = n_1 + n_2$. By induction, there exists $c_1' \leq \mathcal{C}_i c_1$, $c_2' \leq \mathcal{C}_i c_2$, $n_1' \geq n_1$ and $n_2' \geq n_2$ such that

  $$i \vdash^* r_1 : G \Rightarrow^{c_1'}_{Nn_1'} G'', i \vdash^* r_2 : G'' \Rightarrow^{c_2'}_{Nn_2'} G'$$

  By defining $c' = c_1' + c_2'$, $n' = n_1' + n_2'$ we thus as desired obtain $i \vdash^* r : G \Rightarrow^{c'}_{Nn'} G'$. And

  $$c' \leq \mathcal{C}_i(c_1) + \mathcal{C}_i(c_2) = \mathcal{C}_i c,\ n' \geq n_1 + n_2 = n$$

□

By repeated application of lemma 4.6.1, we find

Corollary 4.6.2 Suppose $i \vdash^* r : G \Rightarrow^{c_i}_{Nn_i} G'$, $i > 1$. Then there exists c_1,n_1 such that

$$1 \vdash^* r : G \Rightarrow^{c_1}_{Nn_1} G'.$$

where $c_1 \leq \mathcal{C}_1 \ldots \mathcal{C}_{i-1} c_i$, $n_1 \geq n_i$. □

The partial correctness/speedup theorem(s)

We are now ready for a main theorem, which can be read as follows: suppose G at level i *by an arbitrary strategy* reduces to a normal form. Then G at level 1 *by a normal order strategy* will reduce to *an equivalent* normal form; and the cost of working at level 1 does not exceed the cost of working at level i by more than a factor $\mathcal{C}_1 \ldots \mathcal{C}_{i-1}$.

Theorem 4.6.3 Let G be singlelabeled with result node $n0$. Suppose for $i > 1$ we have

$$i \vdash^* r : G \Rightarrow^{c_i}_{Nn_i} G',\ G' \text{ in well-typed normal form.}$$

Then there exists r', G'' and c_1 such that

$$1 \vdash^* r' : G \Rightarrow^{c_1}_{Nc_1} G'' \text{ where}$$

- G'' is in well-typed normal form, and $\text{Val}_{G'}(r(n0)) = \text{Val}_{G''}(r'(n0))$;
- $n_i \leq c_1 \leq \mathcal{C}_1 \ldots \mathcal{C}_{i-1} \cdot c_i$.

□

PROOF: By corollary 4.6.2 we find that $1 \vdash^* r : G \Rightarrow^{c'}_{Nn'} G'$ where $c' \leq \mathcal{C}_1 \ldots \mathcal{C}_{i-1} c_i$, $n' \geq n_i$. By theorem 4.4.14, there exists G'' in well-typed normal form, reduction r' and c_1 with $n' \leq c_1 \leq c'$ such that $1 \vdash^* r' : G \Rightarrow^{c_1}_{Nc_1} G''$. That $\mathrm{Val}_{G'}(r(n0)) = \mathrm{Val}_{G''}(r'(n0))$ follows from theorem 4.4.11. □

Theorem 4.6.3 is formulated relative to a *fixed* multilevel system (i.e. a fixed set of rules): working *within* this multilevel system one can gain a constant factor only. But given a graph G such that $1 \vdash^* r : G \Rightarrow^{c}_{Nn} G'$ with G' in normal form it will of course always be possible to construct a multilevel system (even a 2-level system) such that $2 \vdash^* r : G \Rightarrow^{1}_{Nn} G'$ – just store the above level 1 transition as a level 1-rule! However, by doing so we have just transferred the cost from "run time" to "rule generation time".

This motivates why we now formulate a speedup bound which does *not* depend on the actual multilevel system (only on the number of levels employed), and which takes "rule generation time" into account:

Theorem 4.6.4 In theorem 4.6.3, we have

$$\mathcal{TT}_{i-1} + c_i \geq i\sqrt[i]{c_1}$$

□

Here the left hand side can be interpreted[10] as the total cost associated with working at level i, and c_1 can be interpreted as the total cost associated with working at level 1 – thus there is justification for the following

Essential Result 4.6.5 *By having an upper bound on the number of levels employed in a multilevel system, one at most gains a* polynomial *speedup.*

PROOF: (of theorem 4.6.4) We have $c_1 \leq \mathcal{C}_1 \ldots \mathcal{C}_{i-1} c_i$, and hence (as $\mathcal{C}_i \leq \mathcal{T}_i$)

$$i \cdot \sqrt[i]{c_1} \leq i \cdot \sqrt[i]{\mathcal{C}_1 \ldots \mathcal{C}_{i-1} c_i} \leq i \cdot \sqrt[i]{\mathcal{T}_1 \ldots \mathcal{T}_{i-1} c_i}$$

Thus the theorem will follow if we can show

$$i \cdot \sqrt[i]{\mathcal{T}_1 \ldots \mathcal{T}_{i-1} c_i} \leq \mathcal{TT}_{i-1} + c_i$$

which amounts to showing

$$i^i \mathcal{T}_1 \ldots \mathcal{T}_{i-1} c_i \leq (\mathcal{T}_1 + \ldots + \mathcal{T}_{i-1} + c_i)^i$$

[10]Wlog. we can assume that a program is run once only, as if it is to be run on several arguments these can be supplied simultaneously.

But this is an instance of the inequality

$$i^i(n_1 \ldots n_i) \leq (n_1 + \ldots + n_i)^i, \text{ all } n_j \geq 0 \tag{4.14}$$

the validity of which follows from the two observations below:

- if $n_1 = \ldots = n_i (= n)$, then (4.14) reads

  $$i^i \cdot n^i \leq (i \cdot n)^i$$

  which certainly holds (with $=$ instead of $\leq$).

- for fixed value of $n_1 + \ldots + n_i$, $n_1 \ldots n_i$ assumes its maximum value when $n_1 = \ldots = n_i$. This is an easy consequence of the observation below, which trivially holds:

  Given n,n' and d, with $0 \leq d \leq n \leq n'$. Then $n \cdot n' \geq (n - d) \cdot (n' + d)$.

□

4.6.1 Total correctness

Theorem 4.6.3 showed that working at higher levels always will be *partially correct*, in the sense that every result could have been achieved at level 1 too. Now we are aiming at conditions for *total correctness*, the meaning of this term being

1. if reduction of G at level i gets "stuck", then it also gets stuck at level 1;
2. if reduction of G at level i "loops", then it also loops at level 1.

Concerning 1, it is easily seen (by combining corollary 4.6.2 and theorem 4.4.14) that the following holds:

Corollary 4.6.6 If $i \vdash^* \_ : G \Rightarrow_{\bar{N}\_} G'$, with G singlelabeled and with G' in normal form but *not* in well-typed normal form, then there exists G'' in normal form but *not* in well-typed normal form such that (for some c) $1 \vdash^* \_ : G \Rightarrow^c_{Nc} G''$. □

On the other hand, a configuration may be "stuck at level i" even if it *does* contain a redex a with $\mathcal{D}(a) \geq 1$ – this will happen if

- one is not allowed to use (all) level 0 rules, when working at level i *and*
- the set of rules one is allowed to use is not "complete".

We do not wish to formulate conditions for a set of rules to be "complete", as such a treatment will depend heavily on the concrete multilevel system – hence we from now on solely focus upon condition 2, i.e. that "looping at level i implies looping at level 1".

The discussion back in section 2.1.2 suggests that "all rules should represent some computation step", so obviously it would be a bad idea if we had $(\mathrm{id}_G : G \Rightarrow^0_{N0} G) \in \mathcal{R}_i$ for some i. However, it is not enough that all rules represent *some* computation step – they should also represent a *useful* computation step. In our formalism (which has been partly designed for this purpose!) this can be coded up in the theorem below which says "if one, when working at level i, only uses *either* level i' rules ($1 \leq i' < i$) representing at least one normal order step *or* a level 0 rule the redex of which is needed; then total correctness is ensured".

Theorem 4.6.7 Given $i > 1$. Assume we have the following (restricted) definition of when $i \vdash r : G \Rightarrow_{Nn} G'$ holds: $(G', r, \_)$ shall be the pushout of $(r_1 + \mathrm{id}_{G_2}, s)$ where s is a specialization from $G_1 + G_2$ to G, and where *either*

1. $(r_1 : G_1 \Rightarrow_{a_1} G_1') \in \mathcal{R}_0$ with $\mathcal{D}(s(\mathsf{in}_1(a_1))) \geq 1$ *or*
2. $(r_1 : G_1 \Rightarrow^{c_1}_{Nn_1} G_1') \in \mathcal{R}_{i'}$ for some $i' < i$ with $n_1 \geq 1$.

Now suppose that G_0 (singlelabeled) is such that for all $k \geq 0$ there exist G_k and n_k such that

$i \vdash^* \_ : G_0 \Rightarrow^k_{Nn_k} G_k$

i.e. "G_0 loops at level i by some strategy". Then G_0 loops at level 1 by a normal order strategy. □

Proof: Let k be given. It is immediate from the assumptions of the theorem that $n_k \geq k$. By corollary 4.6.2 we find that there exists $n_k' \geq n_k (\geq k)$ such that

$1 \vdash^* \_ : G_0 \Rightarrow_{\bar{N}n_k'} G_k$

Now apply theorem 4.4.16. □

It may not be quite obvious how the above theorem applies to concrete multilevel systems. In section 4.7.2, examples will be given to clarify this issue.

Not surprisingly, the same assumptions guarantee that "we do not risk a slowdown by working at level i":

Theorem 4.6.8 Let the assumptions about which transitions are made at level i be as in theorem 4.6.7. Now suppose (with G singlelabeled)

$$i \vdash^* \_ : G \Rightarrow^c_{Nn} G', \text{ with } G' \text{ in normal form.}$$

Then there exists G'' in normal form and $c_1 \geq c$ such that

$$1 \vdash^* \_ : G \Rightarrow^{c_1}_{Nc_1} G''$$

□

Proof: From the assumptions we find that $n \geq c$. By corollary 4.6.2, we find that there exists $n_1 \geq n$ such that $1 \vdash^* \_ : G \Rightarrow_{\bar{N}n_1} G'$. By theorem 4.4.14, we find G'' in normal form and $c_1 \geq n_1$ such that $1 \vdash^* \_ : G \Rightarrow^{c_1}_{Nc_1} G''$; hence the claim. □

The above theorem gives a sufficient condition for "the speedup factor being at least 1". One may ask whether we in general can give conditions for "the speedup factor being at least k". This does not seem quite easy – of course, a natural requirement would be that if one uses a rule $(\_ : \_ \Rightarrow_{\bar{N}n} \_) \in \mathcal{R}_{i'}$, $1 \leq i' < i$ then $n \geq k$. However, excessive use of level 0 rules will make the speedup factor closer to 1 than to k – and we do not want to exclude the possibility of using level 0 rules, as target programs should be allowed to use operators like +!

4.7 Applications of the theory

We now examine how the results from section 4.6 apply to some concrete multilevel systems, in particular those from section 2.1. At the end of the section, we briefly discuss our complexity measure.

4.7.1 Memoization (tabulation)

The fibonacci function is represented by four level 0 rules, two of which are depicted in figure 4.15 (the rule corresponding to *fib*(1) = 1 and the "error rule" are omitted).

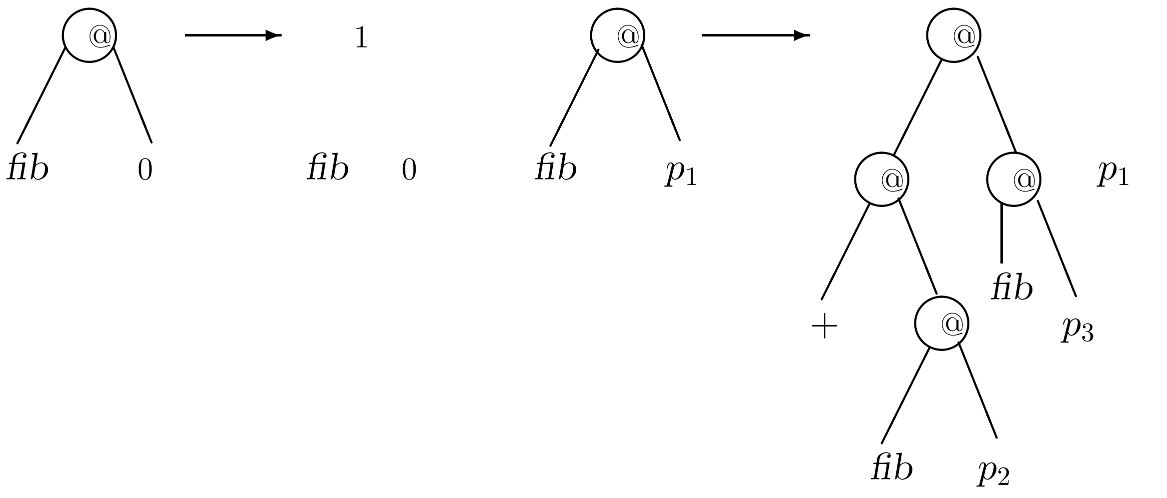


Figure 4.15: Level 0 rules for the fibonacci function.

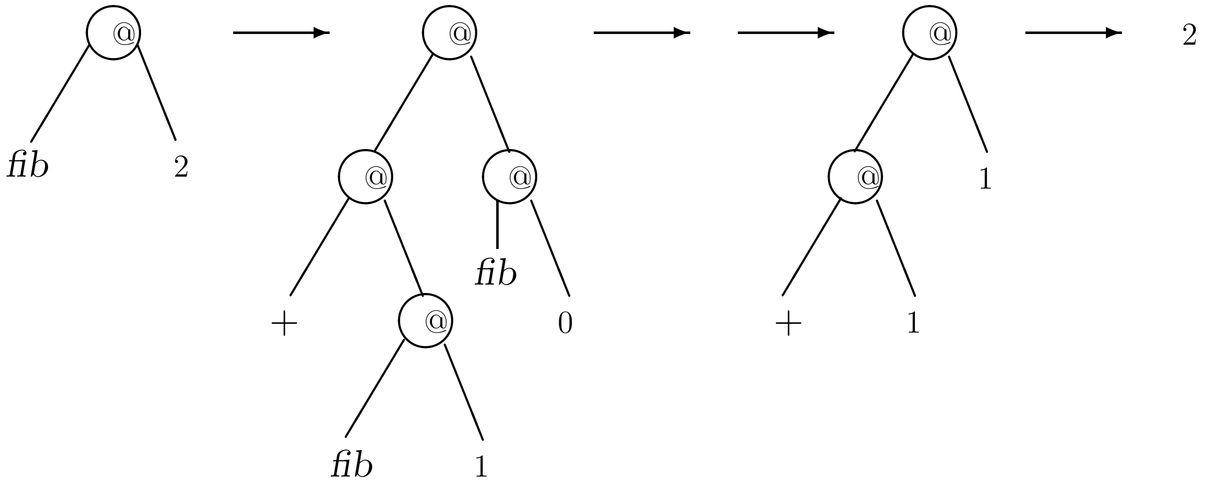


Figure 4.16: Deriving a level 1 rule for *fib*.

In the second rule, the left hand side (right hand side) is equipped with set of labeling functions $\mathcal{L}$ ($\mathcal{L}'$) given by

$$\mathcal{L} = \{l_i | i \geq 2\}, \mathcal{L}' = \{l'_i | i \geq 2\}$$

where $l_i(p_1) = i$; and $l'_i(p_1) = i$, $l'_i(p_2) = i - 1$, $l'_i(p_3) = i - 2$.

Using four level 1 transition steps, we can then generate a level 1 rule (corresponding to $fib(2) = 2$) as depicted in figure 4.16 – as usual, some nodes are implicitly garbage collected.

In general we find $\mathcal{C}_i = 4$, $\mathcal{T}_i = 4$ (as only one rule at each level is present) and $\mathcal{TT}_i = 4i$. Hence the cost of evaluating $fib(n)$ using memoization (or tabulation) is roughly $4n$ (as about n levels are used). On the other hand, the cost of evaluating $fib(n)$ at level 1 is exponential, say about a^n for some a.

The above fits with theorem 4.6.4, which predicts that the cost of working at level n cannot be less than $n\sqrt[n]{a^n}$, i.e. not less than an. So in this example, we (apart from a constant factor) get "as much speedup as we can expect".

The lesson to be learned is (not surprisingly) that one can gain an *exponential* speedup by means of memoization.

4.7.2 Unfold/(fold) transformations

We now reconsider example 2.1.2, where for instance *f(f([a,a]))* evaluates to [*c,c*][11]. Consequently, we expect to have the level 2 reductions depicted in figure 4.17 – the numbers drawn beside some nodes denote the value of the demand function. Let G be the graph on the left hand side of figure 4.17, let $n0$ be its "topmost" node, let G' be the graph on the right hand side and let r be the reduction implicit present in the figure (defined by composition of the three depicted reductions) – then $r(n0)$ will be the topmost node of G'. The demand function has been chosen such that $n0$ is a (the) result node of G (and hence $r(n0)$ is a result node of G') – we employ that $Nd(f) = \{1\}$. In short, we can write

$$2 \vdash^* r : G \Rightarrow^3_{N\_} G' \tag{4.15}$$

That this actually holds is due to $\mathcal{R}_1$ containing four rules (corresponding to those depicted page 24, point 1), one of which (the one corresponding to *f(f(a ::x))* $\rightarrow$ *c ::f(f(x))*) is derived in figure 4.18.

Let G_1 be the graph on the left hand side of figure 4.18, and let G'_1 be the graph on the right hand side. We have

$$1 \vdash^* \_ : G_1 \Rightarrow^2_{N2} G'_1 \tag{4.16}$$

since both redices reduced are labeled with demand 1.

It is important to note that there really *is* a specialization from G_1 to G (so rule (4.16) really *can* be used at level 2) – if the demand functions were chosen in a different way, `INC` might not have been satisfied.

Now let us see how the theorems from section 4.6 apply – we have $\mathcal{C}_1 = 2$, $\mathcal{T}_1 = 4 \cdot 2 = 8$ and $\mathcal{TT}_1 = 8$.

1. Theorem 4.6.3 (applied to (4.15)) states that $1 \vdash^* r' : G \Rightarrow^c_{N\_} G''$, where $c \leq \mathcal{C}_1 \cdot 3 = 6$ and where $\text{Val}_{G'}(r(n0)) = \text{Val}_{G''}(r'(n0))$. Actually, it is quite easy to see that $c = 6$ (and $r' = r$, $G'' = G'$). Hence

[11] We use the standard convention that $[\, v_1, \ldots, v_n \,]$ denotes $(v_1{::}(v_2{::} \ldots (v_n{::}[\,]) \ldots))$.

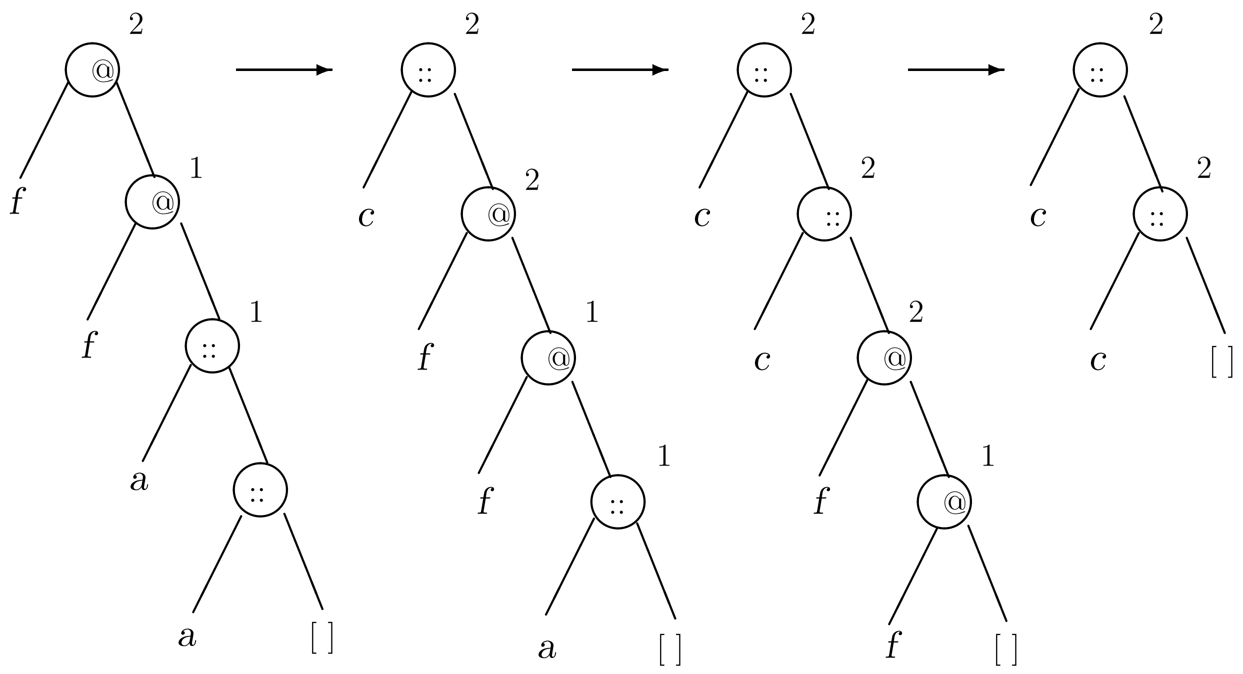


Figure 4.17: Evaluating *f(f([a,a]))* at level 2.

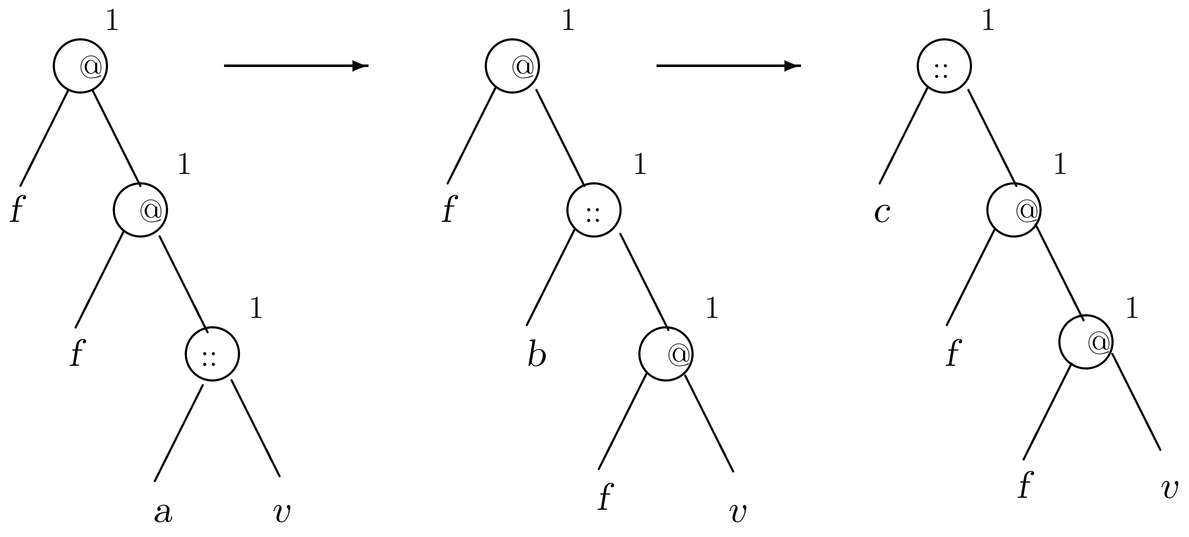


Figure 4.18: Deriving a level 1 rule for *f(f(x))*

we (in our model!) gain a speedup of a factor two, formalizing the intuition presented page 24, point 2.

2. Theorem 4.6.7 states that the transformation (as expected!) is correct (i.e. the domain of termination is not decreased), since all level 1 rules applied (as e.g. (4.16)) represent at least one normal order step (in fact, two).

Total correctness of unfold/(fold) transformations

We now elaborate further on clarifying the implications of theorem 4.6.7, by giving a (toy) example of the unfold/fold technique and then show how this translates into our model. Consider the source program

f(x) $=$ *7*

g(y) $=$ *g(y)*

After making the eureka definitions

h1(x) $=$ *f(g(x))*

h2(x) $=$ *g(f(x))*

one can come up with two transformations:

1. start with *h1(x)*, unfold into *f(g(x))*, unfold *g* yielding *f(g(x))* and then fold back into *h1(x)* – thus deriving the target program *h1(x)* = *h1(x)*.
2. start with *h2(x)*, unfold into *g(f(x))*, unfold *g* yielding *g(f(x))* and then fold back into *h2(x)* – thus deriving the target program *h2(x)* = *h2(x)*.

The new definitions of *h1* and *h2* loop on any input. As the old definition of *h1* terminates (using a normal order strategy) on any input (with result 7), transformation 1 is not correct. On the other hand, as the old definition of *h2* loops on any input, transformation 2 is to be considered correct.

Intuitively, the reason why transformation 2 is correct is that *g* is in a *needed* position when unfolded – on the other hand, in transformation 1 *g* is *not* in a needed position when unfolded.

Now let us encode the above into our framework: here the application of *h1* (*h2*) on (say) 8 is represented as the two graphs G_1 and G_2 depicted

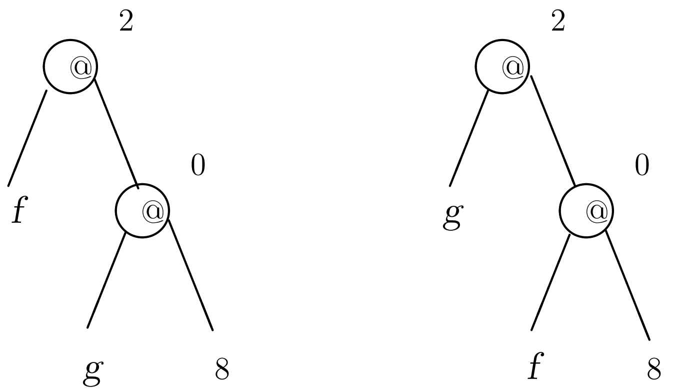


Figure 4.19: $G_1 =$ *f(g(8))*, $G_2 =$ *g(f(8))*.

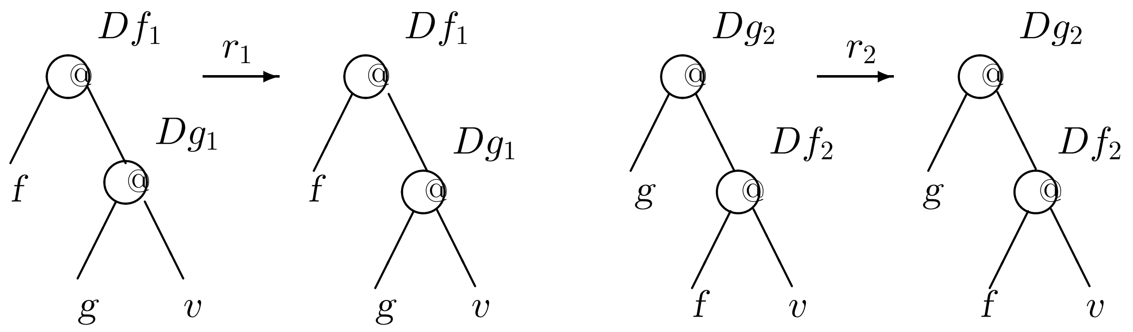


Figure 4.20: Transformation 1 and transformation 2.

in figure 4.19, both having their topmost node as result node (we have $Nd(f) = Nd(g) = \emptyset$). At level 1, G_1 reduces to a normal form (which will still contain the redex *g(8)*; this however has demand 0) in one step. At level 1, G_2 loops (by any strategy).

Transformation 1 and 2 are represented by the two level 1 rules r_1 and r_2, depicted in figure 4.20. Let us discuss how to choose the demands Df_1, Dg_1, Df_2 and Dg_2 (r_1 and r_2 will be valid level 1 transitions no matter what we choose).

First observe that in order for these rules to be applicable when evaluating G_1/G_2 at level 2, we must require $Dg_1 = 0$ and $Df_2 = 0$ (as specializations must satisfy `INC`). On the other hand, we can choose Df_1 and Dg_2 freely.

By assigning Dg_2 the value 1, r_2 represents one normal order step – and hence theorem 4.6.7 says that transformation 2 is correct, as was to be expected.

On the other hand (as Dg_1 has been assigned the value 0) r_1 represents zero normal order steps, and hence theorem 4.6.7 cannot be applied – no wonder, since transformation 1 does *not* preserve termination properties.

4.7.3 Partial evaluation

As hinted at in section 2.1.3, we consider (define) partial evaluation as a 2-level system where the level 1 rules are of form

$$1 \vdash^* r : G \Rightarrow^c_{Nn} G', c \geq 1$$

Here G contains redex a where $\mathit{En}(a, f)$ holds (f is the function being "specialized"), and moreover it must hold that

- for all n in G we have $n \preceq a$ (so the "call of" f does not appear in any "context");
- all other active nodes in G are partial applications (so the arguments to f are virtual or passive – possibly multilabeled).

That we demand $c \geq 1$ (and therefore also had to require a to be a redex) corresponds to the following intuition: first the call of f is replaced by the body of the definition of f (where the known arguments are substituted in); then we do zero or more "simplification steps" (it will be natural to "unfold as far as possible", i.e. demand G' to be in normal form).

Total correctness

The target program resulting from partial evaluation will always have the same termination properties as the source program (if both are evaluated under a normal order strategy). To see this, we have to check that the condition of theorem 4.6.7 is satisfied. This will be the case if $n \geq 1$ (we use the terminology from above), which will hold if $\mathcal{D}(a) = 1$. And provided one during evaluation of the target program only attempts to "reduce redices with demand ≥ 1" (which in effect means that the target program is evaluated under a normal order strategy), such rules are still applicable.

Speedup considerations

Theorem 4.6.3 in effect says that there exists a constant K such that if the source program terminates in c_1 steps and the target program terminates in c_2 steps, then $Kc_2 \geq c_1$. So apparently "one by partial evaluation gains a constant speedup only". However, it is important to note that this K depends on

1. which arguments are static and which are dynamic;
2. the actual value of the static arguments;
3. the partial evaluation strategy used (i.e. whether we "unfold as far as possible").

Of course, one can use theorem 4.6.4 to show that $c_2 + \mathcal{TT}_1 \geq 2\sqrt{c_1}$ ($\mathcal{TT}_1$ should be interpreted as "the cost of producing the target program"), so we can state that "one by partial evaluation gains a polynomial (quadratic) speedup at most".

Now consider the special (but common! [JSS89]) case where an interpreter *int* for a language L (*int* taking two arguments: an L program p and some input i to p) is partially evaluated wrt. a particular L program P. The resulting target program T can be considered equivalent to P, in the sense that T applied to some input I yields the same result as *int* applied to (P, I) – and according to theorem 4.6.3 one never gains more than a factor K by using T instead of using *int*, this K being *independent* of I.

On the other hand, *in practice* K is often independent of P too – that is, K depends on the interpreter only and is often termed "the interpretation overhead". It should be emphasized that this notion of a "fixed" interpretation overhead is *not* supported by our theorems. There are good reasons for this, as it is easy to construct "interpreters" where the corresponding interpretation overhead can become arbitrarily large. However, for many "realistic" interpreters the interpretation overhead in fact is independent of the program being interpreted, loosely speaking due to the fact that each construct in the language being interpreted gives rise to a run-time action.

4.7.4 Discussion of complexity measures

Given a source program s (coded as a set of level 0 rules) and a target program t (coded as an i-level system). Let C be a complexity measure, such that

- $C_s(d)$ is the cost of running s on input d;
- $C_t(d)$ is the cost of running t on d;
- $C(t)$ is the cost of generating t.

So far in this section, we have interpreted theorem 4.6.3 as saying

> there exists a constant $K_{s,t}$ (dependent on s and t) such that for all d,
>
> $$C_s(d) \leq K_{s,t}C_t(d) \tag{4.17}$$

Also, we have interpreted theorem 4.6.4 as saying

> there exists a constant K_s (possibly dependent on s but *not* on t) such that for all d,
>
> $$C(t) + C_t(d) \geq K_s\sqrt[i]{C_s(d)} \tag{4.18}$$

These interpretations are in fact valid (with $K_{s,t} = \mathcal{C}_1 \ldots \mathcal{C}_{i-1}$; $K_s = i$) *provided* $C = C^0$, where C^0 is the complexity measure which assigns unit cost to each function call regardless of

- how difficult it is to "lookup the code" of the function;
- how much "unification" has to be done before the function can be applied.

A very natural question is now: do (4.17) and (4.18) still hold when C is a more "realistic" measure?

When answering this question, we do *not* want to go into details concerning what constitutes a realistic measure – initially we just point out that complexity theory is an area with an enormous number of pitfalls, including:

- If integer operations (addition, subtraction, multiplication, division and equality test) are assigned unit cost, the model "collapses" in the sense that e.g. complete factorization of a number n can be done in time $O(\log n)^2$ [Knu81, p. 398].
- Consider the well-known *Towers of Hanoi* problem (see e.g. [Har87]), which in a logic language can be solved by the program

  hanoi(1,A,B,C,[mv(A,B) | L]-L).

  hanoi(s(N),A,B,C,L-S) :- hanoi(N,A,C,B,L1-S1), hanoi(N,C,B,A,L2-S2),
  append(L1-S1,[mv(A,B) | L2]-S2,L-S).

  append(L1-L2,L2-L3,L1-L3).

where we use the technique of *difference lists* (see e.g. [SS86]) to make it possible to append lists by a single unification.

By using the technique of tabulation, a query *hanoi(n,A,B,C,Moves)* can now be solved using a *number of predicate calls* which is *linear* in n – just build up facts as follows:

hanoi(s(1),A,B,C,[mv(A,C),mv(A,B),mv(C,B) | L]-L)

hanoi(s(s(1)),A,B,C, [mv(A,B),...,mv(A,B) | L]-L)

...

However, as the *size* of the facts themselves grow *exponentially* we do *not* reduce complexity.

- In complexity theory, all functions on a finite set are deemed constant. Thus an intuitively clear statement like “this chess program uses time exponential in the lookahead (the depth of the explored part of the game tree)” makes no sense, as chess is a finite game.

- In classical complexity models, constants do not matter – that is, if X can be recognized in time $f(n)$ then for any $c > 0$ X also can be recognized in time $c \cdot f(n)$. For e.g. Turing machines, the trick is to encode a sequence of symbols into one single symbol[12]. However, recently Neil Jones (as reported in [Nie92, p. 17]) came up with a model, closer to “computational practice”, where constants *do* matter.

After this digression, we return to the question of whether (4.17) and (4.18) can be expected to hold for a realistic complexity measure C – only *very* informal arguments are used!

First observe that we can expect $C(t)$ to be “much larger” than $C^0(t)$, even in the absence of “cheating” like the *Towers of Hanoi* example just mentioned. This reflects the fact that “generation of rules” is rather time consuming due to bookkeeping etc. Hence, we can expect (4.18) to be “even more valid”!

Next take a look at (4.17). This inequality will still hold, provided we can find a constant K such that the cost of making a function call in the

[12]This somewhat conflicts with Turing’s philosophical motivations for the design of his machine [Tur36], namely that there is a bound on the number of symbols a human computer can distinguish between at one glance.

source program is less than K times the cost of making a function call in the target program. To be concrete, consider the Ackerman function from example 2.1.3. If one measures the cost of a function call as the number of arguments to the function, we can there use $K = 2$ – the generalization is immediate …!

4.7.5 Graph representation vs. term representation

From now on, we will (for ease of notation) very seldom explicitly code expressions, programs etc. (represented as terms) into graphs, as the translation back and forth is rather straightforward – variables in terms are coded as virtual nodes in graphs, sharing in graphs is represented by means of `where` abstractions in terms, etc. In particular, we shall often "reason on term level".

A very natural question arises: is this "sound" and "complete", i.e. is "graph rewriting" and "term rewriting" able to "simulate" each other? A similar problem is addressed in [BvEG$^+$87], where it is shown that under certain weak assumptions graph rewriting is a sound and complete implementation of term rewriting. This result does not immediately carry over to our framework, nevertheless it indicates that our somewhat sloppy approach can be justified formally – to do so, however, would be a rather cumbersome task.

4.8 How to get more than a constant speedup

Superficially read, theorem 4.6.3 says that "by doing program transformation within the unfold/fold framework, one can at most gain a constant speedup". This is, of course, *not* a valid interpretation – one must closely examine the assumptions implicitly present in the theory. By doing so we are able to factor out some features, the presence of which enables *more* than a constant speedup:

4.8.1 A non-optimal level 1 evaluation order

Recall that theorem 4.6.3 says that there exists a constant K such that if G at level i (by an arbitrary strategy) reduces to a normal form in c_i steps, then G at level 1 *by a normal order strategy* reduces to normal form in c_1 steps where $c_1 \leq Kc_i$. However, it is *not* claimed that G at level 1 *by an arbitrary strategy* reduces to normal form in a number of steps $\leq Kc_i$.

Thus theorem 4.6.3 does not (immediately) apply to a language with a *strict semantics* (call-by-value) – it is very easy to come up with a counterexample: consider the program

$f(x) = 7$

$g(x) = \varepsilon(x)$

$h(x) = f(g(x))$

where $\varepsilon(x)$ is an expression taking time exponential in (the size of) x. Hence also $h(x)$ will – in a strict semantics – take exponential time. However, by a single unfolding (where a redex is "discarded") we get the target program

$h(x) = 7$

so now $h(x)$ runs in constant time – an exponential speedup has been achieved! Moreover, if $\varepsilon(x)$ is a nonterminating expression we gain an "infinite speedup", that is the domain of termination is *increased*.

It is worth mentioning already now that this source of speedup also occurs (and is perhaps much more common) within the world of logic programming. See section 8.0.1 for a further discussion and examples.

The partial evaluator SIMILIX [BD91] treats a strict language (SCHEME), and it is a design principle that the target program must exhibit exactly the same observable runtime behavior (i.e. same side-effects, same result) as the source program. Hence one cannot allow the domain of termination to be increased, so care is taken to ensure that redices are *not* discarded during transformation (by inserting `let`-expressions). In section 4.10 we briefly sketch how to model this kind of approach.

4.8.2 Introducing sharing during transformation

We have just seen that an exponential speedup can be achieved if one during *transformation* of the source program is allowed to use an evaluation order not permitted during *execution* of the source program. Similarly, an exponential speedup may be achieved if one during transformation is allowed to *identify* some expressions (by means of `where`-abstractions) which one is not permitted to identify during execution. A classical example of this is how one transforms the fibonacci function from being exponential into being linear [BD77]:

First we introduce the function $g(x) = (fib(x), fib(x+1))$ – this is the *tupling strategy*, cf. [Pet84]. For $x = 0$ we by unfoldings alone get

$g(0) \ = (1,1)$

For $x \geq 1$, we perform the transformation sequence

$g(x) \rightarrow (fib(x), fib(x+1)) \rightarrow (fib(x), (fib(x) + fib(x\text{-}1))) \rightarrow$
$\quad (v,(v+u))$ `where` $(u,v) = (fib(x\text{-}1), fib(x))$

and after folding back into g we have thus derived the target function

$g(x) \ = (v,(v+u))$ `where` $(u,v) = g(x\text{-}1)$, if $x \geq 1$.

and now g is clearly linear (and we can compute *fib* using g).

In figure 4.21 and figure 4.22 it is shown how this transformation process is expressed within our graph model. First *assume* that we have the "level 1 rule" depicted in figure 4.21, where G, G' and G'' are equipped with the sets of labeling functions $\mathcal{L}$, $\mathcal{L}'$ and $\mathcal{L}''$ given by

$$\mathcal{L} = \{l_i | i \geq 1\}, \mathcal{L}' = \{l'_i | i \geq 1\}, \mathcal{L}'' = \{l''_i | i \geq 1\},$$

Here we have $l_i(p_1) = l'_i(p_1) = i$, $l_i(p_2) = i + 1$, $l'_i(p_3) = l''_i(p_3) = i$ and $l'_i(p_4) = l''_i(p_4) = i - 1$. Moreover, we have $r_1(a_1) = a'_1$, $r_1(a_2) = a'_2$, $r_2(a'_1) = a''_1$, $r_2(a'_2) = a''_2$ and hence with $r = r_1 \star r_2$ also $r(a_1) = a''_1$, $r(a_2) = a''_2$.

By means of this rule (and the rule corresponding to $g(0) = (1,1)$) $fib(n)$ can be evaluated using $O(n)$ steps – the first two steps in such an evaluation (for $n = 8$) are sketched in figure 4.22. We see that for any $i < n$, only one "copy" of $fib(i)$ exists – hence no sharing is lost.

However, figure 4.21 does *not* denote a level 1 transition – clearly r_1 is, but r_2 is not. If we had an axiom stating

$$1 \vdash^* r : G \Rightarrow^0_{N0} G' \text{ if } r \text{ respects all nodes and is surjective} \qquad (4.19)$$

then it would hold that $1 \vdash^* r_2 : G' \Rightarrow^0_{N0} G''$ – since p_1 and p_3 are labeled identically, we can allow $r_2(p_1) = r_2(p_3)(= p_3)$ and therefore also allow $r_2(a'_1) = r_2(a'_3)$.

To summarize: if we allow (4.19) during *transformation* of the source program but do not allow (4.19) during *execution* of the source program (except for the special case where $r = \text{id\_}$) we are able to get an exponential speedup.

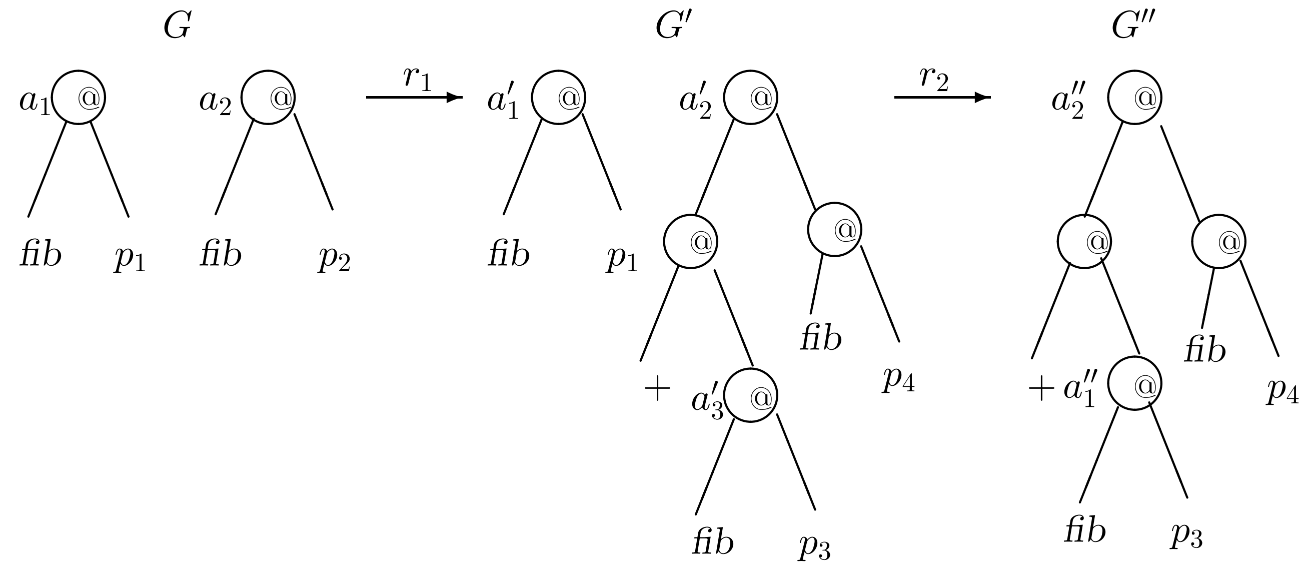


Figure 4.21: Introducing sharing.

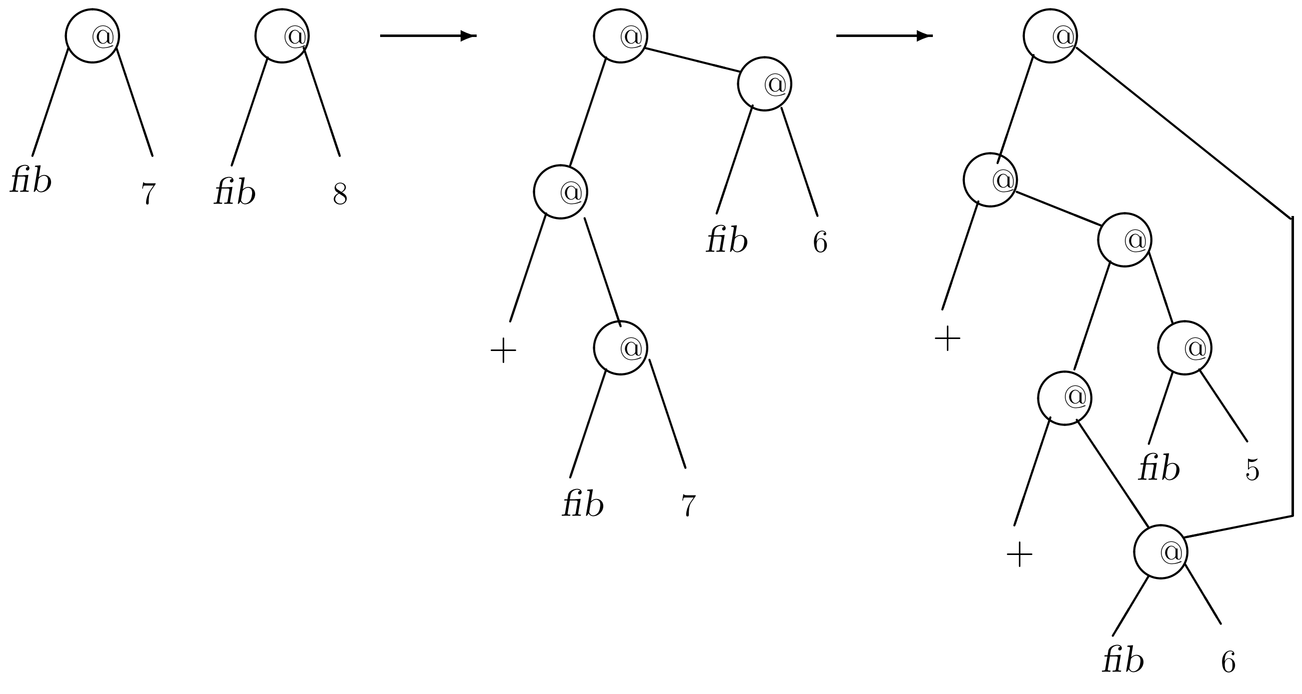


Figure 4.22: Linear evaluation of *fib*.

4.8.3 Proving laws by induction

A very useful and common technique for getting significant speedups is to prove and use certain *algebraic identities* during transformation. In this section we shall see an example of this: exploiting that the operation appending two lists is associative, a list reversing program is transformed from being quadratic into being linear (as in [BD77]).

The source program is depicted below, where ++ (appending two lists) for notational convenience is an infix operator.

rev([ ]) = *[ ]*

rev(v ::x) = *rev(x)* ++ [*v*]

[ ]++ *y* = *y*

(v ::x) ++ *y* = *v ::(x* ++ *y)*

We shall assume that all lists in question are finite. In the following, let $|x|$ be the length of a list x and let $T(e)$ be the number of steps needed for evaluating the expression e using the source program.

Trivially, $T(x\text{++ } y) = |x| + 1$. Concerning $T(rev(x))$, we therefore have the equations

$$\begin{aligned} T(rev([\,])) &= 1, \\ T(rev(v{::}x)) &= 1 + T(rev(x)) + |rev(x)| + 1 = T(rev(x)) + |x| + 2 \end{aligned}$$

and by an easy induction we see that

$$T(rev(x)) = \frac{(|x|+1)(|x|+2)}{2} \tag{4.20}$$

so *rev* is quadratic (as expected).

It turns out to be useful to prove that ++ is associative, i.e. that

$$(x\text{++ } y)\text{++ } z = x\text{++ } (y\text{++ } z) \tag{4.21}$$

for all lists x, y and z. In fact, at the same time we shall prove that

$$T((x\text{++ } y)\text{++ } z) = T(x\text{++ } (y\text{++ } z)) + |x| \tag{4.22}$$

i.e. the right hand side of (4.21) can be *arbitrarily* more efficient than the left hand side – this is the reason why one by replacing the left hand side with the right hand side may get *more than* a constant speedup.

The proof will be conducted by induction on $|x|$ – for $x = [\,]$, the claims are trivial. For $x = (v{::}x')$, the claims follow from the induction hypothesis (applied to x') together with the derivation sequences

$$\begin{array}{lcccc} ((v{::}x') {+\!+}\ y) {+\!+}\ z & \rightarrow & (v{::}(x' {+\!+}\ y)) {+\!+}\ z & \rightarrow & v{::}((x' {+\!+}\ y) {+\!+}\ z) \\ (v{::}x') {+\!+}\ (y {+\!+}\ z) & \rightarrow & & & v{::}(x' {+\!+}\ (y {+\!+}\ z)) \end{array}$$

Now consider expressions[13] of form $rev(x) {+\!+}\ a$ (which is a generalization of the expression $rev(x) {+\!+}\ [\,\mathrm{v}\,]$ met in the source program). For $x = [\,]$ we have

$$rev([\,]) {+\!+}\ a \rightarrow [\,] {+\!+}\ a \rightarrow a$$

and for $x = (v{::}x')$ we have – by one unfolding, one application of the law (4.21) and two more unfoldings –

$$\begin{array}{ll} & rev(v{::}x') {+\!+}\ a \rightarrow (rev(x') {+\!+}\ [\,\mathrm{v}\,]) {+\!+}\ a \\ = & rev(x') {+\!+}\ ([\,\mathrm{v}\,] {+\!+}\ a) \rightarrow^2 rev(x') {+\!+}\ (v{::}a) \end{array} \tag{4.23}$$

By considering *rev1(x,a)* as an abbreviation of *rev(x)* ++ *a*, we have thus derived the target program

rev1([ ],a) $= a$

*rev1((*v *::x),a)* $=$ *rev1(x,(*v *::a))*

by means of which list reversal can be computed in *linear time* (using $rev(x) = rev1(x, [\,])$).

Let us finally elaborate a bit on the exact nature of the speedup gained. Consider the target program execution sequence

$$\begin{array}{lll} & rev([\,v_1 \ldots v_n\,]) \rightarrow rev1([\,v_1 \ldots v_n\,], [\,]) & (4.24) \\ \rightarrow^{i-1} & rev1([\,v_i \ldots v_n\,], [\,v_{i-1} \ldots v_1\,]) \rightarrow & \\ & rev1([\,v_{i+1} \ldots v_n\,], [\,v_i \ldots v_1\,]) & (4.25) \\ \rightarrow^{n-i} & rev1([\,], [\,v_n \ldots v_1\,]) \rightarrow [\,v_n \ldots v_1\,] & (4.26) \end{array}$$

and let us examine how much “progress wrt. the source program” each of those steps represent:

1. (4.24) corresponds to a *negative* progress of size $n+1$ (as it takes $n+1$ steps to rewrite $rev([\,v_1 \ldots v_n\,]) {+\!+}\ [\,]$ into $rev([\,v_1 \ldots v_n\,])$).

[13] a plays the role of an “accumulating parameter”.

2. (4.25), i.e. the "typical step", represents a progress of size $3 + |[\,v_{i+1} \ldots v_n\,]| = n - i + 3$ – this follows from (4.23) combined with (4.22).

3. (4.26) represents a progress of 2 (as it takes two step to rewrite $rev([\,]) +\!+ \, [\,v_n \ldots v_1\,]$ into $[\,v_n \ldots v_1\,]$).

Hence the total progress amounts to

$$\begin{aligned} & -(n+1) + \sum_{i=1}^{n}(n-i+3) + 2 = -n - 1 + n^2 - \sum_{i=1}^{n}(i) + 3n + 2 \\ = \; & n^2 - \frac{n(n+1)}{2} + 2n + 1 = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2} \end{aligned}$$

so now we have derived (4.20) in an alternative way – luckily (!) with the same result.

4.8.4 How to make really big speedups ...

One of the most remarkable successes of the unfold/fold framework is the transformation of the fibonacci function from exponential time into *logarithmic* time, done in [PB82].

Of course it had been known for a long time that it is possible to compute fibonacci numbers in logarithmic time, e.g. by successive squaring of matrices – the virtue of [PB82] is that it is shown how to get from the inefficient program into its efficient counterpart *in a systematic way.*

We now reconstruct the transformation in [PB82], with the aim of exposing the causes of speedup explicitly – at the price of giving no hints at *how* one can come up with such a transformation. Essentially, it is a three-stage process:

1. first make a clever eureka definition;

2. then prove a very useful identity by induction, as in section 4.8.3;

3. finally use the tupling strategy to improve sharing properties, as in section 4.8.2.

Concerning 1, the trick is to *generalize fib* and come up with the definition

$$\begin{aligned} & G(a,b,0) = a \\ & G(a,b,1) = b \\ & G(a,b,n) = G(a,b,n\text{-}1) + G(a,b,n\text{-}2), \text{ if } n \geq 2. \end{aligned}$$

Clearly *fib(n) = G(1,1,n)*.

Concerning 2, we want to prove

$$\begin{aligned}\forall n, k \geq 0 : G(a,b,n+k) &= G(1,0,k)G(a,b,n)\\ &+ G(0,1,k)G(a,b,n+1) \end{aligned} \tag{4.27}$$

This can be done by induction in k (in [PB82] this induction is only done implicitly): if $k = 0$ (4.27) reads

$$\forall n \geq 0 : G(a,b,n) = 1 \cdot G(a,b,n) + 0 \cdot G(a,b,n+1)$$

which certainly holds; and if $k = 1$ (4.27) reads

$$\forall n \geq 0 : G(a,b,n+1) = 0 \cdot G(a,b,n) + 1 \cdot G(a,b,n+1)$$

which also is true. For $k \geq 2$, we have (with the second equality sign due to the induction hypothesis)

$$\begin{aligned} & G(a,b,n+k)\\ = \ & G(a,b,n+(k-1)) + G(a,b,n+(k-2))\\ = \ & G(1,0,k-1)G(a,b,n) + G(0,1,k-1)G(a,b,n+1) +\\ & G(1,0,k-2)G(a,b,n) + G(0,1,k-2)G(a,b,n+1)\\ = \ & (G(1,0,k-1) + G(1,0,k-2))G(a,b,n) +\\ & (G(0,1,k-1) + G(0,1,k-2))G(a,b,n+1)\\ = \ & G(1,0,k)G(a,b,n) + G(0,1,k)G(a,b,n+1)\end{aligned}$$

As a corollary to (4.27), we get

$$\begin{aligned} G(1,0,1+k) &= G(1,0,k)G(1,0,1) + G(0,1,k)G(1,0,2)\\ &= 0 + G(0,1,k)(0+1) = G(0,1,k)\end{aligned} \tag{4.28}$$

$$\begin{aligned} G(0,1,1+k) &= G(1,0,k)G(0,1,1) + G(0,1,k)G(0,1,2)\\ &= G(1,0,k) + G(0,1,k)\end{aligned} \tag{4.29}$$

From (4.27) we can derive a program which is roughly quadratic (as we have $T(2n) \approx 4 \cdot T(n)$). In order to do better, we embark on point 3: it turns out to be a good idea to define

t(k) = (G(1,0,k),G(0,1,k),G(1,1,k),G(1,1,k+1))

We clearly have

$$t(0) = (1, 0, 1, 1)$$

and if $t(k) = (p, q, r, s)$ we have (by means of (4.27), (4.28) and (4.29))

$$\begin{aligned}
t(2k) &= (G(1,0,k)G(1,0,k) + G(0,1,k)G(1,0,k+1),\\
&\quad G(1,0,k)G(0,1,k) + G(0,1,k)G(0,1,k+1),\\
&\quad G(1,0,k)G(1,1,k) + G(0,1,k)G(1,1,k+1),\\
&\quad G(1,0,k)G(1,1,k+1) + G(0,1,k)G(1,1,k+2))\\
&= (p^2 + q^2, pq + q(p+q), pr + qs, ps + q(r+s))
\end{aligned}$$

Also, still with $t(k) = (p, q, r, s)$, we have

$$\begin{aligned}
t(2k+1) &= (G(1,0,k)G(1,0,k+1) + G(0,1,k)G(1,0,k+2),\\
&\quad G(1,0,k)G(0,1,k+1) + G(0,1,k)G(0,1,k+2),\\
&\quad G(1,0,k)G(1,1,k+1) + G(0,1,k)G(1,1,k+2),\\
&\quad G(1,0,k)G(1,1,k+2) + G(0,1,k)G(1,1,k+3))\\
&= (pq + q(q+p), p(p+q) + q(p+2q),\\
&\quad ps + q(r+s), p(r+s) + q(2s+r)
\end{aligned}$$

The expressions for $t(2k)$ and $t(2k+1)$ clearly define a function with logarithmic runtime.

4.9 Related work

In this section the work reported so far in this chapter is compared with other approaches from the literature, with special emphasis on papers discussing "bounds of speedup possible" and/or "conditions for total (partial) correctness". We attempt the order of exposition to be chronological, but do not claim the list to be exhaustive.

The seminal paper on unfold/fold transformations [BD77] addresses both the abovementioned issues. Concerning correctness, an informal argument (attributed to Gordon Plotkin) is given for *partial* (but not total) correctness: let I be the set of function symbols, let E_s be the set of equations in the source program, and let $\{s_i | i \in I\}$ be the (domain-theoretic) functions thus defined. Clearly the s_i's satisfy E_s. By manipulating E_s, we end up with a set of equations, E_t, constituting the target program. As the unfold/fold rules clearly are "sound", any function satisfying E_s

also satisfies E_t. Now, domain theory tells us that $\{t_i | i \in I\}$, the "new meaning" of the function symbols, is to be found as the *least* functions satisfying E_t. From the s_i's satisfying E_t we therefore infer $t_i \leq s_i$ – but this just amounts to partial correctness. Some remarks:

- From section 4.8.1 we recall that partial correctness does *not* necessarily hold if a strict semantics is used. Accordingly, the above argument implicitly presupposes a non-strict semantics (corresponding to a normal order evaluation strategy) in order for "equational reasoning" to be valid – otherwise, it would not be sound to e.g. infer $f(g(x)) = 7$ from an equation $f(x) = 7$.

  Interestingly enough, Appendix 1 in [BD77] presupposes a *call-by-value* strategy: a program is given which (by such a strategy) is rather inefficient, as large data structures are built but only small parts are used. It is said that even though the program originally was put forward as an argument for the concept of *coroutines*[14], a similar economy in computation can be gained by the unfold/fold technique. Their development corresponds to the observation in section 4.8.1: by discarding some redices during transformation one gains a (potentially arbitrarily big) speedup wrt. a call-by-value semantics.

- The abovementioned argument for *partial correctness* does not give any clue concerning how to argue for *total correctness* of a given transformation – of course it will suffice if we can prove that any function satisfying E_t also satisfies E_s[15], but we would certainly like a condition more based on the "syntactic structure" of the transformation. It seems very hard to come up with such conditions using the framework of denotational semantics, cf. my claims p. 12.

Concerning speed-up issues, [BD77, p. 48] informally reasons as follows:

1. unfolding leaves efficiency unchanged;
2. application of laws, as well as introduction of `where`-abstractions, are potential sources of efficiency improvements.

[14]Essentially giving rise to the same flow of control as lazy evaluation.

[15]As pointed out by David Sands this in general seems unfeasible, as an essential ingredient of program transformation is to "forget" some information.

3. folding (at least) preserves efficiency (and therefore also guarantees total correctness, as the introduction of looping surely does not preserve efficiency), provided "one folds with an argument which is lower in some well-founded ordering than the argument in the equation being transformed"[16].

1 does not hold in our model as we measure complexity in terms of the number of unfoldings (while [BD77] (p. 65) measures complexity in terms of the number of arithmetic operations). 2 is just the core insight of section 4.8.3 and section 4.8.2. There is no counterpart to 3 in our work: we give conditions on the "function symbol level" (i.e. on which functions are unfolded), not on the "function argument level" – hence there is no need for devising a well-founded ordering relation!

[Kot80] describes a framework for proving programs to be equivalent, using two techniques:

1. The so-called *McCarthy method* – p_1 and p_2 are equivalent if there exists p_3 such that p_1 and p_2 both transform to p_3 using the unfold/fold method (also using some predefined laws).

2. The so-called *second order replacement method* – if two terms t_1 and t_2 are equivalent, one can substitute t_2 for t_1.

The (in our view) most interesting claim of the paper is that technique 1 alone is *not* complete, in the sense that there exist programs which are equivalent but cannot be proven so by means of 1 (of course, when technique 2 is added completeness is trivial!) The reason is stated p. 67:

> The proof [that the McCarthy method is incomplete] is a direct consequence of the fact that fold/unfold method performs only linear equivalences ...

Here the definition of a "linear equivalence" (p. 66) between source program s and target program t amounts to saying that there exists a linear function f such that if expression e in n steps reduces to e' when evaluated using t, then e in $f(n)$ steps reduces to e'' when evaluated using s where e'' is "more defined than" e' (for instance we could have $e'' = c(c(x))$ and $e' = c(g(x))$ with c being a constructor symbol and g being a function

[16]This is often considered *the* recipe for how to ensure total correctness, as e.g. in [Hen87, p. 183].

symbol). The steps are assumed to be "parallel outermost steps" (resembling normal order evaluation). No proofs (or references to such) are given of the abovementioned "fact" – but the content is rather close to theorem 4.6.3...

In [Kot85] (identical to the technical report [Kot82]) the problem of assuring total correctness is addressed (but, alas, again all theorems are given without proofs or references to proofs). The language treated is a first-order functional language (without branching). The setup is that first a number of unfoldings are made; then some predefined laws are applied; and finally a number of foldings are made. Some conditions for total correctness are given, e.g. proposition 3 and theorem 2 (p. 428) – these essentially say that a transformation is safe if the number of unfoldings exceeds the number of foldings. It is worth noting that chapter 8, which addresses the problem of total correctness for a logic language, is built upon a generalized version of this intuition.

[Sch80] introduces (using a first-order functional language with call-by-value semantics) the notion of *expression procedures*, a variant over the unfold/fold framework – as in our model, folding is viewed as an abbreviation. Some syntactic requirements on transformations are put forward guaranteeing total correctness – for instance, one must only substitute within a "strict" context: from a rule *k(x)* $\rightarrow$ *k(x)* it would be wrong to deduce the rule *If p(x) h(x) k(x)* $\rightarrow$ *If p(x) h(x) k(x)*, as this will introduce a loop which (in the case of *p(x)* evaluating to `true`) was not present before. The correctness proof works by exploiting that an expression e terminates iff there exists a well-founded ordering $\prec$ on expressions derivable from e such that if e in one step rewrites to e' then $e' \prec e$.

In [JSS89] one finds the following remarks:

> Program transformation is concerned with rather radical changes to a program's structure, so the final program may have properties very different from those of the original one. A common goal for instance is to change a program's running time as a function of input size, often from exponential to polynomial or from polynomial to linear.
>
> We have argued that partial evaluation can achieve order-of-magnitude *linear* speedups (e.g. of target programs over interpreters) but it seems unlikely that partial evaluation can

> yield non-linear speedups in general. One reason is that partial evaluation uses only a single transformation technique, essentially a generalization of well-known compiler optimizations.
>
> So the goals of partial evaluation are in a sense more modest and, we think, achievable by simpler methods than those of program transformation in general.

This intuition has, so to speak, been formalized in section 4.8.

In [Red89] an alternative approach to unfold/fold transformations is presented, using concepts from the theory of term rewriting systems[17]. For instance, if one to the usual rules for the fibonacci function *fib* adds the "eureka definition" $<fib(n),fib(n+1)> \rightarrow g(n)$ then the computation of "critical pairs" in some sense simulates the usual unfold/fold process.

In [Hon91] it is shown that unfoldings, foldings and introduction of `where`-abstractions do not (modulo a constant) change the "inherent complexity" of a function, where the inherent complexity is the number of steps needed to evaluate the function on an ideal parallel machine (i.e. no communication overhead etc.) – for instance, the "exponential" fibonacci function (example 2.1.1) and the "linear" fibonacci function (section 4.8.2) both have *linear* inherent complexity. A denotational approach is attempted, making it possible to express the inherent complexity as the "number of fixed point unfoldings needed" – however, some operational reasoning nevertheless (as to be expected) sneaks into the theory. In section 4.10 we will sketch how to extend our model to encompass inherent complexity.

[Mar91] investigates a class of term rewriting systems where terms are labeled – thereby implicitly defining DAGS but *not* cyclic graphs in general. By focusing upon the derivations where all redices with same label are reduced simultaneously, graph reduction is mimiced. Now, loosely speaking, a "call-by-need derivation" is one where only needed redices are reduced, a redex being "needed" if it can never disappear (except when reduced). It is proved that "call-by-need" derivations are optimal – analogous to[18] our theorem 4.4.14. In general it can only be known "after-

[17]The relationship between these two frameworks in general seems rather overlooked – other attempts to narrow the gap include [Bel91]. A substantial difference is that the term rewriting community often restricts attention to *terminating* reduction sequences.

[18]Another result in this direction is presented in [Yos93], where a weak λ-calculus (i.e. no reductions under a λ) with explicit environments (so the Church-Rosser property holds) is

wards" (from a normalizing derivation) what constitutes a needed redex, but for some subclasses of term rewriting systems (including supercombinators *without* pattern matching) we know that the "leftmost-outermost" redex will be needed[19].

Note that the purpose of the demand function employed in our model is to be able to capture "neededness" in a syntactic way (i.e. without looking at some future derivations), in some sense generalizing the concept of "leftmost-outermost" redex to combinators *with* pattern matching.

An interesting approach to reasoning about cost is given in [San90, chap. 4], where bisimulation techniques (well-known from e.g. [Mil89]) are used for formalizing that "two expressions compute the same answer using the same amount of time" – to be more precise we say that R is a cost simulation iff, whenever $e_1 \; R \; e_2$, the following holds: if e_1 using k_1 steps reduces to a q_1 in "head normal form", there exists q_2 (in head normal form) such that e_2 in k_2 steps reduces to q_2, such that $k_1 = k_2$ and such that *either* q_1 and q_2 are identical "atomic" values *or* q_1 and q_2 are composite data structures where the arguments are related pairwise by R.

An immediate generalization of the above approach would be to relax the requirement $k_1 = k_2$ and demand only $k_1 \leq c \cdot k_2$, with c a constant (depending solely on R). Then it might be possible to reason about when a concrete transformation yields a constant speedup only.

[Hof92] considers *jungle evaluation*, a kind of term rewriting systems based on graph grammars. Instead of *overwriting* a node with the result of "evaluating it", as in our model, a *pointer* is drawn from the node to the result.

The rewriting system uses four kinds of rules:

1. *evaluation rules*, which correspond to replacing a function call by the body of the function;

2. *tabulation rules*, which are evaluation rules that in addition make preparations for storing the result of the function call;

treated. In this setting, it is proved that the leftmost reduction strategy is optimal.

[19]Similar work in this direction includes [Sta80], which investigates general (acyclic) graph reductions and whose Result 3 states that "sound contractions" (reducing "sound nodes") are "optimal contractions", a "sound node" loosely speaking being one the "ancestors" of which are never reduced before the node itself is – the paper then gives criteria for being sound. Also, [BKKS87] investigates the issue of detecting redices in a λ-expression the reduction of which is needed in order to arrive at (head) normal form.

3. *lookup rules*, which model the retrieval of a previously stored result;
4. *folding rules*, which increase sharing by identifying nodes "with identical children".

Rule 2 and 3 account for "lazy memoization" (i.e. on equality of pointers, cf. the discussion in section 2.1.1); by also employing 4 one can get "full memoization" (i.e. on structural equality) – the latter is implemented in [Kah92] (cf. section 3.5).

In our model, rule 1 corresponds to "use of level 0 rules"; rule 2 and rule 3 are implicitly present in the multilevel framework; and rule 4 correspond to the rule (4.19) proposed in section 4.8.2.

[Hof92, theorem 4.13] states a correctness result, saying that an evaluation using 2,3 and 4 can be simulated using 1 and 4 – this in some sense corresponds to our partial correctness theorem 4.6.3.

4.10 Possible extensions to the model

In this section we briefly outline some directions in which our model could be extended.

Inherent complexity

In order to model "inherent complexity" as used in [Hon91] (cf. section 4.9), we allow one "step" to perform several reductions *in parallel.* To be more precise, for $i \geq 1$ we say that $i \vdash r : G \Rightarrow_{N\_} G'$ (we do not care about the number of normal order steps) provided there exist level i' rules ($i' < i$) $r_1 \dots r_p$ ($p \geq 0$), graph G_1 and specialization s such that $(G, r, \_)$ is the pushout of $((r_1 + \dots + r_p + \mathrm{id}_{G_1}), s)$. Moreover, we have the inference rules

$$\frac{i \vdash r : G \Rightarrow_{N\_} G'}{i \vdash^* r : G \Rightarrow^1_{N\_} G'} \tag{4.30}$$

$$i \vdash^* r : G \Rightarrow^0_{N\_} G' \text{ if } r \text{ respects all nodes and is surjective} \tag{4.31}$$

$$\frac{i \vdash^* r_1 : G_1 \Rightarrow^{c_1}_{N\_} G_2, i \vdash^* r_2 : G_2 \Rightarrow^{c_2}_{N\_} G_3}{i \vdash^* r_1 \star r_2 : G_1 \Rightarrow^{(c_1+c_2)}_{N\_} G_3}$$

with rule (4.31) corresponding to (4.19). Clearly, we have $i \vdash^* \mathrm{id}_G : G \Rightarrow^c_{N\_} G$ for any c (use $p = 0$ above).

Similar to what we did in section 4.5 we can show that if $i \vdash^* r : G \Rightarrow^c_{N\_} G'$, s is a specialization from G to G_1 and $(G'_1, r_1, \_)$ is the pushout of (r, s), then also $i \vdash^* r_1 : G_1 \Rightarrow^c_{N\_} G'_1$.

Also, if $i \vdash r_1 : G_1 \Rightarrow_{N\_} G'_1$ and $i \vdash r_2 : G_2 \Rightarrow_{N\_} G'_2$ then $i \vdash r_1{+}r_2 : G_1{+}G_2 \Rightarrow_{N\_} G'_1{+}G'_2$. Then it is easy to show that if

$$i \vdash^* r_1 : G_1 \Rightarrow^{c_1}_{N\_} G'_1 \text{ and } i \vdash^* r_2 : G_2 \Rightarrow^{c_2}_{N\_} G'_2$$

then $i \vdash^* r_1{+}r_2 : G_1{+}G_2 \Rightarrow^c_{N\_} G'_1{+}G'_2$, where $c = \max(c_1, c_2)$.

Lemma 4.6.1 then still holds: if $i + 1 \vdash^* r : G \Rightarrow^c_{N\_} G'$ with $i \geq 1$, then there exists $c' \leq \mathcal{C}_i \cdot c$ such that $i \vdash^* r : G \Rightarrow^{c'}_{N\_} G'$.

In order to simulate the results of [Hon91], we (as one cannot use (4.31) during "execution of the source program") have to show that if $1 \vdash^* r : G \Rightarrow^c_{N\_} G'$ then also $1 \vdash^* r : G \Rightarrow^c_{N\_} G''$ (with G'' "similar" to G'), where the latter derivation does *not* use rule (4.31). It will clearly be sufficient if we can show that an application of (4.31) followed by an application of (4.30), can be replaced by an application of (4.30) followed by an application of (4.31). But this should not be too difficult.

Modeling call-by-value

Recall from section 4.8.1 that one by discarding redices during transformation may increase the domain of termination for a strict language (thus violating partial correctness); hence it may be of interest to come up with conditions ensuring that all transformations are "strict".

We define a predicate B on the set of virtual nodes, with the intuitive interpretation that if $\mathsf{B}(v)$ holds then v cannot be a placeholder for a graph containing active nodes.

Then define $n \in \mathcal{B}$, a predicate on the set of all nodes with the intuitive interpretation that if $n \in \mathcal{B}$ then n can never appear in a context where there are active nodes "below" n. $n \in \mathcal{B}$ will hold iff it is not possible to deduce $\vdash n \notin \mathcal{B}$, using the following inference system:

$$\frac{\text{not } \mathsf{B}(v)}{\vdash v \notin \mathcal{B}} \quad \frac{\vdash \mathcal{S}(p, i) \notin \mathcal{B}}{\vdash p \notin \mathcal{B}} \quad \vdash a \notin \mathcal{B}$$

Now require all morphisms m in question (reductions, isomorphisms etc.) to satisfy that

$$\text{if } \mathsf{B}(v) \text{ then } m(v) \in \mathcal{B}'. \tag{4.32}$$

As all such m in addition respect all passive nodes, we can deduce that $n \in \mathcal{B}$ implies $m(n) \in \mathcal{B}'$ – this just amounts to proving that $\vdash m(n) \notin \mathcal{B}'$ implies $\vdash n \notin \mathcal{B}$, and is done by induction in the proof tree of the former derivation.

In addition, we require isomorphisms and reductions m to satisfy: $\mathsf{B}'(v')$ iff there exists v with $m(v) = v'$ such that $\mathsf{B}(v)$.

Then the development in section 4.1 carries through – the only non-trivial point is to ensure that the pushout exists. Using the terminology from section 4.1.5, equip G' with a predicate B' such that $\mathsf{B}'(V')$ iff there exists V in G with $r'(V) = V'$ such that $\mathsf{B}(V)$ holds. We now only need to check that s' satisfies (4.32): assume $\mathsf{B}'(v')$, then there exists v with $\mathsf{B}(v)$ such that $r(v) = v'$. As s satisfies (4.32), $s(v) \in \mathcal{B}$; and as r' (by definition) satisfies (4.32), $r'(s(v)) \in \mathcal{B}'$ – but $r'(s(v)) = s'(r(v)) = s'(v')$, hence the claim.

Now demand all virtual nodes, appearing in level 0 rules, to satisfy $\mathsf{B}(v)$ – then it is impossible to do a step which is not call-by-value. And it is easy to derive the following theorem, expressing partial correctness as well as "only a constant speedup is possible":

Theorem 4.10.1 Given G. Suppose for $i > 1$ we have

$i \vdash^* r : G \Rightarrow^{c_i}_{N\_} G'$

Then there exists $c_1 \leq \mathcal{C}_1 \dots \mathcal{C}_{i-1} \cdot c_i$ such that

$1 \vdash^* r : G \Rightarrow^{c_1}_{N\_} G'$

□

Modeling garbage collection

Recall the remark on page 44: if reductions only need to be *partial* mappings, i.e. we allow $r(n) = \perp$, then garbage collection is modeled. However, one has to ensure that one does not throw away something which is needed in another context. Therefore we define a predicate E, the intuitive interpretation of $E(n)$ being that "some larger context needs n".

We then must demand reductions r to satisfy: if $E(n)$ then $r(n) \neq \perp$; and $E'(n')$ should hold iff there exists n with $E(n)$ and $r(n) = n'$. Also, we must demand specializations s to be total and to satisfy: if $s(n_1) = s(n_2)$ with $n_1 \neq n_2$ then $E(n_1)$ and $E(n_2)$; and if $E'(s(n))$ then also $E(n)$.

Then it will be possible to proceed as before.

A more symmetric notion of reduction

In order to model folding which is *not* an abbreviation (cf. p. 24), it would be convenient if reductions were “less directed” – corresponding to the intuition that folding represents a step in the “wrong” direction. That is, instead of r being a (partial) mapping we would rather like r to be a *relation*. We have not investigated this idea further in the current setting – however, in chapter 8 we shall see a logic model where it is possible to reverse *some* “unfoldings”.

Chapter 5

The Ultimate Sharing Machine

As promised in chapter 3, we now define a *non-deterministic* machine (to be called an USM) for ultimate sharing (the concept presented in section 3.1), using the framework from chapter 4. Further, we prove that under some rather weak and natural conditions the machine is "correct". The machine can, of course, be made (almost) deterministic in several ways – one such way is described in great detail, as it amounts to a top-down version of partial evaluation (cf. section 2.1.3).

5.1 The USM

We suppose a predefined set of level 0 rules, $\mathcal{R}_0$. On the other hand, for $i \geq 1$ the set of level i rules $\mathcal{R}_i$ is initially empty and will gradually be built up while the USM runs. A crucial property of the machine is that the sets $\mathcal{R}_i$ form a "monotone" data structure in the sense that elements are added, but never deleted. In other words, if $(r : G \Rightarrow^c_{Nn} G') \in \mathcal{R}_i$ holds at some time it will hold at any later time. This implies that if $i \vdash^* r : G \Rightarrow^c_{Nn} G'$ holds at some time, it will hold ever after. As usual, we will demand that if $(r : G \Rightarrow^c_{Nn} G') \in \mathcal{R}_i$ then also $i \vdash^* r : G \Rightarrow^c_{Nn} G'$. For ease of exposition, we do not explicitly include the sets $\mathcal{R}_i$ in the configurations, but rather treat the addition of a rule to $\mathcal{R}_i$ as a "side-effect".

The configurations

A configuration of the USM (to be thought of as a stack, with the top being rightmost) takes the form

$$[t_1 l_1 t_2 \ldots t_{k-1} l_{k-1} t_k], k \geq 1 \qquad (5.1)$$

where k is termed the *height* of the configuration. Here each t_j takes the form $(G_j, G'_j, r_j, c_j, n_j, i_j)$ where r_j is a reduction from G_j to G'_j. G'_k is termed *the current goal* of the configuration. The following relation is an invariant of the USM:

$$\forall j \in \{1 \dots k\} : i_j \vdash^* r_j : G_j \Rightarrow^{c_j}_{N n_j} G'_j \tag{5.2}$$

Each l_j takes the form (s_j, H_j) (H_j a graph). We have the invariant:

$$\forall j \in \{1 \dots k-1\} : s_j \text{ is a specialization from } G_{j+1}{+}H_j \text{ to } G'_j \tag{5.3}$$

The intuitive interpretation of the USM being in configuration (5.1) is as follows: initially it started to evaluate G_1; when reaching G'_1 it preferred to look at a G_2 "more general than" G'_1; then it evaluated G_2 until reaching G'_2;...; finally it started evaluating G_k and has now reached G'_k.

Given a graph G we can define $\mathsf{SC}(G)$, the "start configuration given by G", by

$$\mathsf{SC}(G) = [(G, G, \mathrm{id}_G, 0, 0, 1)]$$

which clearly satisfies (5.2) and (5.3). A *terminal configuration* takes the form

$$[(G, G', r, c, n, i)], \text{ with } G' \text{ in well-typed normal form.}$$

The transitions

The USM is able to make three kinds of transitions, each of which is easily seen to preserve the invariants (5.2) and (5.3):

- A `PUSH` step amounts to "looking at a more general graph". To be more precise, we have

$$\begin{array}{ll} & [\dots(G_k, G'_k, r_k, c_k, n_k, i_k)] \\ \rhd & [\dots(G_k, G'_k, r_k, c_k, n_k, i_k), (s, H), (G, G, \mathrm{id}_G, 0, 0, 1)] \end{array}$$

  provided s is a specialization from $G{+}H$ to G'_k. We say that G has been pushed.

- An `UNFOLD` step amounts to "exploiting a previously generated rule" (or a level 0 rule). To be more precise, we have (with r' a reduction from G'_k to G''_k)

$$\begin{array}{ll} & [\dots(G_k, G'_k, r_k, c_k, n_k, i_k)] \\ \rhd & [\dots(G_k, G''_k, r_k{\star}r', c_k+1, n_k+n, \max(i_k, i+1))] \end{array}$$

  provided $(G''_k, r', \_)$ is the pushout of $(r{+}\mathrm{id}_\_, s)$ where *either*

– $(r : \_ \Rightarrow_{Nn}^{\bar{}} \_) \in \mathcal{R}_i$ *or*

– $(r : \_ \Rightarrow_a \_) \in \mathcal{R}_0$, in which case $i = 0$, and $n = 1$ if $\mathcal{D}(s(a)) \geq 1$ ($\mathcal{D}$ being the demand function of G'_k), otherwise $n = 0$.

(To ensure that (5.2) is preserved, we exploit that if $i \vdash^* r : G \Rightarrow_{Nn}^c G'$ then also $i' \vdash^* r : G \Rightarrow_{Nn}^c G'$ for $i' > i$.) An UNFOLD step is said to be *progressing* if $n > 0$.

- A POP step amounts to "stop working at the more general graph; but save the result and use it". To be more precise, we have

$$\begin{array}{rl} & [\ldots(G_k, G'_k, r_k, c_k, n_k, i_k)(s, H)(G, G', r, c, n, i)] \\ \rhd & [\ldots(G_k, G''_k, r_k \star r', c_k + 1, n_k + n, \max(i_k, i + 1))] \end{array}$$

where $(G''_k, r', \_)$ is the pushout of $(r + \mathrm{id}_H, s)$. At the same time we extend $\mathcal{R}_i$ such that $(r : G \Rightarrow_{Nn}^c G') \in \mathcal{R}_i$. A POP step is said to be *progressing* if $n > 0$.

Given a configuration of form

$$[(G_1, G'_1, r_1, c_1, n_1, i_1)\ldots(G_k, G'_k, r_k, c_k, n_k, i_k)] \tag{5.4}$$

we define $\mathcal{I} = \max(i_1 \ldots i_k)$. The following invariant of $\rhd$ is easy to check:

Fact 5.1.1 Suppose $\mathsf{SC}(G) \rhd^*$ a configuration of form (5.4). Then $G = G_1$, $\mathcal{I} \geq 1$, and for $j \geq 1$ we have $\mathcal{R}_j = \emptyset$ iff $j \geq \mathcal{I}$. □

A complexity measure

Recall the definition of $\mathcal{C}_i$, $\mathcal{T}_i$ and $\mathcal{TT}_i$ from section 4.6. Given a configuration of form (5.4), we now define

$$\mathcal{CC} = c_1 + \ldots + c_k + \mathcal{TT}_i, i = \mathcal{I} - 1$$

and when considering an USM execution sequence of the form $C_1 \rhd^* C_2$, $\mathcal{CC}$ refers to the *latter* configuration C_2. Intuitively, $\mathcal{CC}$ is the "work done by the USM so far". This is not an unreasonable interpretation, as shown by

Fact 5.1.2 Starting from a configuration of form $\mathsf{SC}(G)$, we have

$$\mathcal{CC} = |\mathtt{POP}| + |\mathtt{UNFOLD}| + \mathcal{I} - 1 \tag{5.5}$$

(here e.g. |POP| denotes the number of POP steps). □

PROOF:

- Initially, (5.5) reads $0 + 0 = 0 + 0 + 1 - 1$, which is true.
- A `PUSH` step leaves both sides of (5.5) unchanged.
- An `UNFOLD` step adds one to both sides of (5.5) – to see this, observe that the step does not change the value of $\mathcal{I}$: as $\mathcal{R}_i$ is not empty, fact 5.1.1 says that $i < \mathcal{I}$ and hence $i + 1 \leq \mathcal{I}$.
- For a `POP` step, where $\mathcal{R}_i$ is extended such that $(r : G \Rightarrow^{c}_{Nn} G') \in \mathcal{R}_i$, we must distinguish between two cases:
  - If $i < \mathcal{I}$, the left hand side of (5.5) is increased by $-c+1+c = 1$; and the right hand side of (5.5) is increased by 1.
  - If $i = \mathcal{I}$, $\mathcal{I}$ will be increased by one. Hence the left hand side of (5.5) will be increased by $-c+1+c+1 = 2$ (due to the way $\mathcal{T}_i$ is defined in section 4.6); and the right hand side of (5.5) will be increased by 2.

□

Partial correctness and speedup

Theorem 5.1.3 Suppose $\mathsf{SC}(G) \rhd^* [(G, G', r, c, n, i)]$, where G is single-labeled with result node $n0$ and G' is in well-typed normal form. Then there exists r', G'' and c_1 such that

$1 \vdash^* r' : G \Rightarrow^{c_1}_{Nc_1} G''$ where

- G'' is in well-typed normal form, and $\mathrm{Val}_{G'}(r(n0)) = \mathrm{Val}_{G''}(r'(n0))$;
- $c_1 \leq \mathcal{C}_1 \ldots \mathcal{C}_{i-1} \cdot c$;
- $\mathcal{CC} \geq i\sqrt[i]{c_1}$.

□

PROOF: As (5.2) is an invariant of $\rhd$, we have

$i \vdash^* r : G \Rightarrow^{c}_{Nn} G'$

Then apply theorem 4.6.3 and theorem 4.6.4, exploiting $\mathcal{CC} = c + \mathcal{TT}_{i-1}$.
□

Let us briefly digress on "stuck" configurations: a stuck configuration of the USM must be of form $[(G, G', r, c, n, i)]$ (as otherwise we could do a `POP` step). But now G' is in normal form (as otherwise we could do an `UNFOLD` step using a level 0 rule). Hence, as a terminal configuration is not considered stuck, $i \vdash^* r : G \Rightarrow^c_{Nn} G'$ with G' in normal form but not in well-typed normal form. Then apply corollary 4.6.6 to see that G also at level 1 by a normal order strategy ends up in a "stuck" configuration.

Total correctness

Now we embark on giving conditions for total correctness, i.e. for "if the USM loops starting from G, then G loops at level 1 by a normal order strategy".

For a USM configuration (of form (5.4)), we define $\mathcal{N}$ as $n_1 + \ldots + n_k$ – the intuitive interpretation is "the total number of level 1 normal order steps performed by the USM". Observe that progressing `UNFOLD` steps increase $\mathcal{N}$; all other steps leave $\mathcal{N}$ unchanged.

Definition 5.1.4 Let $S_1 \rhd S_2 \rhd \ldots \rhd S_i \rhd \ldots$ be a (possibly infinite) execution sequence of the USM. We say that the sequence is *progressing* provided

1. all `POP` steps are progressing;
2. if the sequence is infinite then it contains an infinite number of steps which are either `POP` steps or progressing `UNFOLD` steps.

□

The intuition behind 1 is that one should not stop computation and save the result before one has done something useful. Condition 2 amounts to saying that there should not exist an infinite sequence containing `PUSH` steps and non-progressing `UNFOLD` steps only, that is one should not repeatedly look at more and more general configurations without doing something useful in between.

It turns out that if a USM carries out progressing execution sequences only, total correctness is guaranteed. To show this, two lemmas are needed:

Lemma 5.1.5 Suppose there is a *progressing* infinite execution sequence of the USM. Then $\mathcal{N}$ will grow towards infinity. □

Proof: Assume $\mathcal{N}$ is bounded (for the sake of getting a contradiction). That is, after some point t the USM never performs a progressing `UNFOLD` step. As the execution sequence is progressing we then (by condition 2) can conclude that the USM performs an infinite number of `POP` steps. Now two possibilities:

- A `PUSH` step is made after point t. As all subsequent `UNFOLD` steps are non-progressing, the next `POP` step (and there is such a step!) will be non-progressing. But this violates condition 1.
- No `PUSH` steps are made after point t. But as the stack cannot shrink to negative size, this is impossible.

□

Lemma 5.1.6 Suppose G (singlelabeled) is such that there exists an execution sequence of the USM, starting from $\mathsf{SC}(G)$, where $\mathcal{N}$ grows towards infinity. Then G loops at level 1 by a normal order strategy. □

Proof: Given n. By assumption, the USM encounters a configuration with $\mathcal{N} \geq n$. Now perform a sequence of `POP` steps (always possible) – as this leaves $\mathcal{N}$ unchanged, we end up with a configuration of form

$[(G_1, G_1', r_1, c_1, i_1, n_1)]$ with $n_1 \geq n$

By fact 5.1.1 $G_1 = G$. As (5.2) holds, we have

$i_1 \vdash^* r_1 : G \Rightarrow_{Nn_1}^{c_1} G_1'$

By corollary 4.6.2, there exists $n_1' \geq n_1 (\geq n)$ such that

$1 \vdash^* r_1 : G \Rightarrow_{\bar{N}n_1'} G_1'$

Now apply theorem 4.4.16. □
By combining lemma 5.1.5 and lemma 5.1.6 we get our "correctness theorem":

Theorem 5.1.7 Suppose G (singlelabeled) is such that there exists an infinite and *progressing* execution sequence of the USM starting from $\mathsf{SC}(G)$. Then G loops at level 1 by a normal order strategy. □

5.2 A machine for partial evaluation

We now outline an (almost) deterministic version of the USM, to be called a PEM (Partial Evaluation Machine), which simulates the effects of partial evaluation.

For ease of exposition, we make the following assumptions (which are immediate to generalize):

- all functions f have three arguments: x_1, x_2 and x_3;
- for all functions f, x_1 is the only argument "needed by f" (cf. the function *Nd* from section 4.3) – i.e. in order to unfold f one needs to know the value of x_1 but not the value of x_2 or x_3;
- the static arguments of f (cf. section 2.1.3) are x_1 (no loss of generality, as in order to do any computation one has to know the value of x_1) and x_2.

Let the current goal of the PEM (cf. page 133) be G'. Now the next action of the PEM is found according to the *priority* list below:

1. *if* the height of the current configuration is 1 *and* G' contains a function application of form $f(\alpha_1, \alpha_2, e_3)$ with demand ≥ 1 *and* there exists a rule in $\mathcal{R}_i$ ($i \geq 1$) with left hand side $f(\alpha_1, \alpha_2, x_3)$, *then* perform an `UNFOLD` step using this rule;
2. *if* the height of the current configuration is 1 *and* G' contains a function application of form $f(\alpha_1, \alpha_2, e_3)$ with demand ≥ 1 *then* perform a `PUSH` step, pushing $f(\alpha_1, \alpha_2, x_3)$ (giving this application demand 1, which is clearly possible);
3. *if* G' contains a function application of form $f(\alpha_1, e_2, e_3)$ with demand ≥ 1 *then* perform an `UNFOLD` step using the corresponding level 0 rule (which will exist);
4. *if* the height of the current configuration is > 1, then perform a `POP` step;
5. *otherwise* halt.

The intuition is as follows: when working at the "top level" (i.e. when executing the "target program"), one should attempt to use specialized functions – if they already exist (1) use them, otherwise (2) start to generate them. Apart from that, unfold as far as possible (3 and 4).

It is easily seen that if/when the PEM stops, the configuration will be of form $[(G, G', \_, \_, \_, \_)]$ where G' does not contain a redex with demand ≥ 1, i.e. G' is in normal form.

An invariant

Fact 5.2.1 Assuming that the PEM starts from $\mathsf{SC}(G)$, the invariant below will hold during execution:

> A configuration either takes the form $[t_1]$ or the form $[t_1 l_1 t_2]$. With t_1 of form $(G, G_1', \_, \_, \_, i)$ we have $i = 1$ or $i = 2$. With t_2 (if it exists) of form $(G_2, G_2', \_, c, n, i)$ we have $i = 1$, $c = n$, and G_2 is of form $f(\alpha_1, \alpha_2, x_3)$ – this application having demand 1. For $i \geq 2$, $\mathcal{R}_i = \emptyset$. If $(\_ : G \Rightarrow^{c}_{Nn} \_) \in \mathcal{R}_1$ then G is of form $f(\alpha_1, \alpha_2, x_3)$ (this application having demand 1), and $c = n > 0$ (hence each `POP` step is progressing). Also, each `UNFOLD` step is progressing.

$\square$

Proof: Mostly a straightforward induction in the number of steps performed by the PEM, using case analysis wrt. 1-4 above. Perhaps the only interesting point is to assure that each `POP` step is progressing – so consider the configuration before a `POP` step:

$$[(G, G_1', \_, \_, \_, \_) l_1 (G_2, G_2', \_, c, n, i)]$$

Suppose $n = 0$. Then by the invariant also $c = 0$, implying $G_2' = G_2$. But then G_2' is a redex with demand 1, hence it is possible to do a step of type 3 – so the PEM will *not* do a `POP` step! $\square$

Correctness

That partial correctness holds is an instance of theorem 5.1.3; now for total correctness: from fact 5.2.1 it is easy to see that any execution sequence of the PEM will be progressing. Hence we can apply theorem 5.1.7 and get

Theorem 5.2.2 Suppose G (singlelabeled) is such that there exists an infinite execution sequence of the PEM starting from $\mathsf{SC}(G)$. Then G loops at level 1 by a normal order strategy. $\Box$

Speedup issues

Concerning the speedup possible by using the PEM, we can employ theorem 5.1.3 where we by fact 5.2.1 have $i = 1$ or $i = 2$. We see that we cannot hope for more than a *quadratic* speedup. As the PEM amounts to a top-down implementation of partial evaluation, this is just a restatement of an observation made already in section 4.7.3.

5.2.1 A larger example

We now illustrate the behavior of the PEM by running it on an interpreter for a small language (called TINY) with syntax

$$e ::= c \mid \mathsf{Input} \mid \mathsf{Plus}(e, e) \mid \mathsf{Rec}(e) \mid \mathsf{If}(e, e, e)$$

Here c ranges over integer constants, Input refers to the (sole) input of the program, $\mathsf{If}(e_1, e_2, e_3)$ branches according to whether e_1 evaluates to zero, and $\mathsf{Rec}(e)$ calls the program recursively with e as input. As an example, consider the TINY program Π below which given input n computes $n/2$ (assuming n is even and positive):

$$\Pi = \mathsf{If}(\mathsf{Input}, 0, \mathsf{Plus}(1, \mathsf{Rec}(\mathsf{Plus}(\mathsf{Input}, (-2)))))$$

We now present an interpreter for TINY with three parameters: the current expression, the whole program and the input – it is straightforward to code this interpreter as a set of level 0 rules (where *Int* and *Branch* both need their first argument).

Int(c,p,d) $=$ *c*

*Int(*Input*,p,d)* $=$ *d*

*Int(*Plus*(e1,e2),p,d)* $=$ *Int(e1,p,d)* $+$ *Int(e2,p,d)*

*Int(*Rec*(e),p,d)* $=$ *Int(p,p,Int(e,p,d))*

*Int(*If*(e1,e2,e3),p,d)* $=$ *Branch((Int(e1,p,d)=0),e2,e3,p,d)*

*Branch(*true*,e2,e3,p,d)* $=$ *Int(e2,p,d)*

*Branch(*false*,e2,e3,p,d)* $=$ *Int(e3,p,d)*

Now let us see what happens when the PEM is run on the expression $\mathit{Int}(\Pi, \Pi, 6)$, where all arguments except the last to *Int* and *Branch* are to be considered static. Some remarks: we do *not* generate specialized versions of $+$ and $=$; and in the following it is easy to check that we only reduce redices with demand ≥ 1.

First the PEM does a `PUSH` in order to look at the expression $\mathit{Int}(\Pi, \Pi, v)$. After doing an `UNFOLD` we arrive at

$$\mathit{Branch}((\mathit{Int}(\mathsf{Input}, \Pi, v) = 0), 0, \Phi, \Pi, v)$$

with Φ denoting the expression $\mathsf{Plus}(1, \mathsf{Rec}(\mathsf{Plus}(\mathsf{Input}, (-2))))$. By one more `UNFOLD` step we arrive at $\mathit{Branch}(v = 0, 0, \Phi, \Pi, v)$. We cannot do further `UNFOLD` steps, but do a `POP` step enabling us to store the level 1 rule

$$1 \vdash^{*} {}_{-} : \mathit{Int}(\Pi, \Pi, v) \Rightarrow_{\bar{N}_{-}} \mathit{Branch}(v = 0, 0, \Phi, \Pi, v) \tag{5.6}$$

and at the top level we now have the expression $\mathit{Branch}(6 = 0, 0, \Phi, \Pi, 6)$. By an `UNFOLD` step we get $\mathit{Branch}(\mathsf{false}, 0, \Phi, \Pi, 6)$, and now we do a `PUSH` and look at the expression $\mathit{Branch}(\mathsf{false}, 0, \Phi, \Pi, v)$. By an `UNFOLD` step, we get $\mathit{Int}(\Phi, \Pi, v)$ which by one more `UNFOLD` step yields

$$\mathit{Int}(1, \Pi, v) + \mathit{Int}(\mathsf{Rec}(\mathsf{Plus}(\mathsf{Input}, (-2))), \Pi, v)$$

By two `UNFOLD` steps we arrive at

$$1 + \mathit{Int}(\Pi, \Pi, \mathit{Int}(\mathsf{Plus}(\mathsf{Input}, (-2)), \Pi, v))$$

and by one more `UNFOLD` step we get

$$\begin{array}{rl} & 1 + \mathit{Branch}(\mathit{Int}(\mathsf{Input}, \Pi, v1) = 0, 0, \Phi, \Pi, v1) \\ \texttt{where} & v1 = \mathit{Int}(\mathsf{Plus}(\mathsf{Input}, (-2)), \Pi, v) \end{array}$$

The `where` abstraction is used in order to reflect that our graph model does not duplicate arguments. By one `UNFOLD` step we get

$$\begin{array}{rl} & 1 + \mathit{Branch}(v1 = 0, 0, \Phi, \Pi, v1) \\ \texttt{where} & v1 = \mathit{Int}(\mathsf{Plus}(\mathsf{Input}, (-2)), \Pi, v) \end{array}$$

which by one more `UNFOLD` step rewrites into

$$\begin{array}{rl} & 1 + \mathit{Branch}(v1 = 0, 0, \Phi, \Pi, v1) \\ \texttt{where} & v1 = \mathit{Int}(\mathsf{Input}, \Pi, v) + \mathit{Int}((-2), \Pi, v) \end{array}$$

and by two `UNFOLD` steps we finally get

$$1 + \mathit{Branch}(v1 = 0, 0, \Phi, \Pi, v1) \texttt{ where } v1 = v + (-2)$$

We cannot do further `UNFOLD` steps, but do a `POP` step enabling us to store the level 1 rule

$$\begin{aligned} 1 \;\; \vdash^* {}_{-} : \;\; & \mathit{Branch}(\mathsf{false}, 0, \Phi, \Pi, v) \Rightarrow_{\bar{N}_{-}} \\ & 1 + \mathit{Branch}(v1 = 0, 0, \Phi, \Pi, v1) \texttt{ where } v1 = v + (-2) \end{aligned} \tag{5.7}$$

and we return to the top level with the expression (as v is to be "bound" to 6)

$$1 + \mathit{Branch}(v1 = 0, 0, \Phi, \Pi, v1) \texttt{ where } v1 = 6 + (-2)$$

By two `UNFOLD` steps, we get

$$1 + \mathit{Branch}(\mathsf{false}, 0, \Phi, \Pi, 4)$$

Fortunately, we can now reuse rule (5.7) and by an `UNFOLD` step we get (with v bound to 4)

$$1 + 1 + \mathit{Branch}(v1 = 0, 0, \Phi, \Pi, v1) \texttt{ where } v1 = 4 + (-2)$$

Still working at top level, we do two `UNFOLD` steps and arrive at

$$1 + 1 + \mathit{Branch}(\mathsf{false}, 0, \Phi, \Pi, 2)$$

Again, we can reuse rule (5.7) and by an `UNFOLD` step get

$$1 + 1 + 1 + \mathit{Branch}(v1 = 0, 0, \Phi, \Pi, v1) \texttt{ where } v1 = 2 + (-2)$$

which by two `UNFOLD` steps rewrites into

$$1 + 1 + 1 + \mathit{Branch}(\mathsf{true}, 0, \Phi, \Pi, 0)$$

Now the PEM performs (even though it does not save any work) a `PUSH` step and looks at the configuration $\mathit{Branch}(\mathsf{true}, 0, \Phi, \Pi, v)$. By one `UNFOLD` step we get $\mathit{Int}(0, \Pi, v)$ and by one more `UNFOLD` step we get 0, enabling us to do a `POP` step and store the level 1 rule

$$1 \vdash^* {}_{-} : \mathit{Branch}(\mathsf{true}, 0, \Phi, \Pi, v) \Rightarrow_{\bar{N}_{-}} 0 \tag{5.8}$$

At the same time we return to top level, with the expression $1+1+1+0$ which by a few `UNFOLD` steps rewrites to 3, the "final result".

5.2.2 The PEM versus (bottom-up) partial evaluation

The PEM employs the strategy to "unfold as far as possible" during specialization. This closely corresponds to the unfolding strategy used by most modern partial evaluators (eg. SIMILIX [BD91]), where the specialization algorithm unfolds until meeting a test which cannot be decided – such "dynamic tests" will then be the *specialization points.* Accordingly, in the example from section 5.2.1 the right hand side of the level 1 rules contains the function *Branch*, but *not* the function *Int* (as the latter represents a *static* test which can be unfolded).

An advantage of the PEM over bottom-up PE is that it preserves termination properties (theorem 5.2.2 ctr. the discussion in section 2.1.3). A disadvantage is that the time used for looking up the appropriate level 1 rule may be prohibitively big – on the other hand, one can do some optimization: instead of storing the rule (5.7) as it is one should rather store it as

$$1 \vdash^* \_ : \mathit{Branch}_1(\mathsf{false}, v) \;\Rightarrow_{\bar{N}\_}\; 1 + \mathit{Branch}_1(v1 = 0, v1) \\ \texttt{where } v1 = v + (-2)$$

where $\mathit{Branch}_1(b, v)$ "denotes" $\mathit{Branch}(b, 0, \Phi, \Pi, v)$. Due to this trick, the system implemented in [AT89] is not unreasonably inefficient. One might also devise a clever hash function to enable quick retrieval of rules, similar to what is done in [Kah92].

5.3 Discussion and related work

An interesting variation of the PEM is to drop the requirement that the height is ≤ 2 – that is, one is allowed to do a PUSH step or to do an UNFOLD step using a rule in $\mathcal{R}_i$, $i \geq 1$ even if the height of the current configuration is > 1. Then $\mathcal{I}$ can grow arbitrarily big, hence an exponential speedup is possible in principle – in chapter 6 we shall see that there actually *exists* a "natural" problem where an exponential speedup can be gained by such an approach. On the other hand, if the example from section 5.2.1 is run with this strategy a lot of superfluous rules will be stored, as a rule will be generated for each subexpression of the program being interpreted (this, on the other hand, may be useful for "incremental compilation" [AT89]).

A system implementing something rather similar to the PEM (with the above extension, but without ultimate sharing in its full generality) is described in [AT89]. The correctness proof of the system is conducted by means of a well-founded ordering, namely the lexicographical ordering of $\aleph^3$. The first component essentially (i.e. when translated into the framework presented here) measures the number of steps needed to reduce an expression at level 1; the second component essentially measures "how many `PUSH` steps are left before we start to do something useful" (cf. our definition of an execution sequence being progressing); and the third component is the "size" of the expression (the need for this comes from the use of a structural operational semantics).

As already mentioned in section 3.5, [Kah92] describes a system where all nodes in the heap are unique. This is a very clean and uniform approach; our approach – with the explicit distinction between the (cheap) sharing provided by the graph reduction mechanism itself and the (expensive) sharing provided by the memoization aspect – is perhaps more flexible.

Chapter 6

Simulating a 2DPDA by Ultimate Sharing

As promised in section 3.4 and section 5.3, we now present a "realistic" program which by means of ultimate sharing can be made to run exponentially faster. The program to be considered is a simulator for *two-way deterministic pushdown automata* (2DPDA) [AHU74, chap. 9]. It caused much surprise when [Coo71] showed that it is always possible to simulate a 2DPDA in linear time (wrt. the length of the input tape), even if the automaton carries out an exponential number of steps – in particular this result gave Donald Knuth inspiration to his fast substring matching algorithm [KMP77, p. 338]. We shall see that the effect of this clever simulation can be acquired using the general concept of ultimate sharing.

This chapter is based on joint work with Jesper Träff which has been reported in [AT92] (but the basic idea dates back to [AT89]). The exposition here is substantially different, as we can build upon the general theory developed in the previous chapters.

We proceed as follows: first we define a 2DPDA, write a (naive) simulator and give an example of an automaton which runs in exponential time. Next we define a deterministic version of the ultimate sharing machine (closely resembling the PEM, with the extensions proposed in section 5.3). We show that it is "correct" to run this machine on our simulator, and that we by doing so obtain *linear* runtime. Finally, we compare with previous work.

6.1 Defining 2DPDAs

A 2DPDA is an automaton operating on a *read-only* tape and on a "push-down store" (i.e. a stack where only the top element can be accessed). The action to be chosen next by the automaton is determined from its current internal state, the current tape symbol and *the top element* of the stack. An action *either* halts the automaton (and announces "accept" or "reject") *or* consists of three subactions: 1) the internal state is (possibly) changed; 2) the "tape head" is (possibly) moved one step to the left or one step to the right[1]; 3) the stack is *either* left as it is *or* the topmost element is removed *or* a symbol is pushed upon the stack.

2DPDAs are interesting as they recognize a large class of languages, encompassing

1. all regular languages – this is immediate since this amounts to the languages recognized by a DFA, and a DFA is a special case of a 2DPDA;

2. all deterministic context-free languages – also this is immediate since these are defined as those which can be recognized by a *one*-way deterministic push-down automaton, this also being a special case of a 2DPDA (where only tape moves to the right are allowed). This class contains [HU79, p. 261] exactly those languages which can be generated by means of $LR(1)$ grammars, this class in turn being equal to the class of languages which can be generated by means of $LR(k)$ grammars.

2DPDAs can even recognize some languages which are not context-free, e.g. $\{a^n b^n c^n | n \geq 0\}$. According to [HU79, p. 124], it is not known whether there exists a context-free language which cannot be recognized by a 2DPDA.

In order to write a simulator for 2DPDAs, we must find a way to code automata and tapes. Our approach will be to have (conceptually!) a set of level 0 rules for each 2DPDA and each tape – thus there will be an infinite number of level 0 rules, cf. page 83. For instance, if the 27'th tape (assuming some enumeration) is of length 4 and is of form *abaa* we

[1]Our development would not be affected if we allowed the head to move an arbitrary number of steps.

shall have the level 0 rules

$$\begin{array}{lll} \textit{tape27}(0) = \dagger_L & \textit{tape27}(1) = a & \textit{tape27}(2) = b \\ \textit{tape27}(3) = a & \textit{tape27}(4) = a & \textit{tape27}(5) = \dagger_R \end{array}$$

where $\dagger_L$ ($\dagger_R$) is a special symbol denoting the left end (right end) of a tape.

In a similar way we can code automata: suppose e.g. that the 17'th automation in state σ upon reading b on the tape and reading ω on top of the stack performs the action: push a ω_1, move 1 step to the left and enter state σ_1. Then there will be a level 0 rule

$$\textit{aut17}(\omega, \sigma, b) = ((\mathsf{PUSH}\ \omega_1), \sigma_1, -1)$$

The symbol PUSH should not be confused with the symbol `PUSH`: the former denotes that the stack *of the 2DPDA* is pushed; the latter that the stack *of the USM* is pushed. We also have symbols POP and LEAVE, denoting that the stack is popped/left unchanged; and have symbols ACCEPT and REJECT with the obvious meaning. Finally, there will be a distinguished state symbol σ_0 and a distinguished stack symbol Z_0: the former denotes the "initial state"; the latter denotes the "bottom of the stack".

To avoid runtime errors, we must make two requirements:

1. We do not have level 0 rules of form $aut_i(\_, \_, \dagger_L) = (\_, \_, -1)$ (in order to stay within the tape). Similarly for $\dagger_R$.

2. We do not have level 0 rules of form $aut_i(Z_0, \_, \_) = (\mathsf{POP}, \_, \_)$ (in order not to pop an empty stack).

We are now in position to present the simulator *sim* which easily can be coded up as level 0 rules (*sim* needs its third argument and *branch* needs its first argument):

sim(aut,tape,(stacktop::stackrest),state,pos) =
    branch(aut(stacktop,state,tape(pos)),aut,tape,stacktop,stackrest,pos)

*branch(((*PUSH *newtop),newstate,move),aut,tape,stacktop,stackrest,pos)* =
    sim(aut,tape,(newtop :: (stacktop :: stackrest)),newstate,pos+move)

*branch((*POP*,newstate,move),aut,tape,stacktop,stackrest,pos)* =
    sim(aut,tape,stackrest,newstate,pos+move)

*branch((*LEAVE*,newstate,move),aut,tape,stacktop,stackrest,pos)* =

sim(aut,tape,(stacktop :: stackrest),newstate,pos+move)

*branch(*ACCEPT*,aut,tape,stacktop,stackrest,pos)* $=$ ACCEPT

*branch(*REJECT*,aut,tape,stacktop,stackrest,pos)* $=$ REJECT

To simulate the actions of 2DPDA α when run on tape τ, proceed as follows: find i and j such that α is represented by the function aut_i, τ is represented by the function $tape_j$. Then evaluate the expression

$$sim(aut_i, tape_j, [\,Z_0\,], \sigma_0, 1) \tag{6.1}$$

It is immediate that the simulator is correct in the following sense: with G of form (6.1) there exists a G' in normal form such that $1 \vdash^* \_ : G \Rightarrow^c_{Nc} G'$ iff α terminates when run on τ (and G' then "codes the result"). This c will be denoted $T(\alpha,\tau)$ and equals $4k$, where k is the number of steps performed by α when run on τ.

6.2 Complexity of the simulator

For a tape τ, $|\tau|$ denotes the length of the tape ($\dagger_L$ and $\dagger_R$ included); for a 2DPDA α, $|\alpha|$ denotes the number of state symbols times the number of stack symbols.

sim has (at least) exponential (worst-case) complexity, as expressed in

Fact 6.2.1 There exists a 2DPDA α_{exp} with the following property: it terminates on all input tapes, but for all N there exists a $n \geq N$ and a τ with $|\tau| = n$, such that $T(\alpha_{\text{exp}}, \tau) \geq 2^n$. □

(We shall now exhibit such an automaton; note that this automaton is the "natural" way to encode an "interesting" general problem and thus cannot be considered "contrived"...)

Proof: Much as in [ANTJ89] we come up with a parametrized construction: for each DFA (with transition function δ) using the binary alphabet $\{0,1\}$, a 2DPDA is constructed which given input tape of form

$$\dagger_L \overbrace{a \dots a}^{n} \dagger_R$$

decides whether there exists a string of n binary digits which are accepted by the DFA. The idea is to first test whether the DFA accepts $00\dots00$, then test whether the DFA accepts $00\ldots01$, then test whether the DFA accepts $00\ldots10$, etc.

For ease of exposition assume that a 2DPDA in *one* step is able to make *two* PUSH actions, and also assume that it is possible in *one* step first to do a POP action, then test on the *resulting* stack top and afterwards perform *yet* a POP action possibly *followed by* some PUSH actions – it is straightforward how to eliminate this "syntactic sugar".

The 2DPDA has two states: `pushzero`, which takes the role as the initial state σ_0, and `next`. The automaton will be designed in such a way that the following invariant holds:

> Suppose the number of a's to the left of the tape head is i ($0 \leq i \leq n$, but the tape head never points at $\dagger_L$). Then the stack is of form (it grows to the right)
>
> $$s_0 d_1 s_1 \ldots s_{j-1} d_j s_j \ldots d_i s_i \qquad (6.2)$$
>
> where each s_j is a state of the DFA and s_0 is the initial state, where each d_j is a bit (i.e. belongs to $\{0,1\}$) and where for all $j \in \{1 \ldots i\}$ we have $\delta(s_{j-1}, d_j) = s_j$. Wlog. we can assume that s_0 does not occur in the range of δ – then we can safely identify s_0 with Z_0, causing (6.2) to hold initially.
>
> Concerning "which bit strings have been tested for acceptance", we have
>
> - When in state `pushzero`, the next string of length n to test for acceptance (wrt. the DFA) is $d_1 \ldots d_i\ 0\ \ldots 0$ – all binary strings (of length n) strictly less than (wrt. the standard ordering) that string have already been rejected.
> - When in state `next`, all binary strings (of length n) less than or equal $d_1 \ldots d_i\ 1\ \ldots 1$ have already been rejected.

Next for the full definition of the 2DPDA:

- When in state `pushzero`, the next action of the 2DPDA is found according to the priority list below:

    1. Suppose the current tape symbol is $\dagger_R$ and the current stack symbol is an accepting state of the DFA. Then return ACCEPT (the invariant tells us that $i = n$, and that $d_1 \ldots d_i$ is an accepting string).

2. Suppose the current tape symbol is $\dagger_R$ and the current stack symbol is not an accepting state of the DFA. Then transfer control to state `next` (the invariant tells us that $i = n$ and that all strings strictly less than $d_1 \dots d_i$ have been rejected; as the DFA does not accept $d_1 \dots d_i$ the invariant for `next` follows).
3. Let the current stack symbol be s (from the invariant we know that s is a state of the DFA), and let $s' = \delta(s, 0)$. Then push a 0 followed by s' on the stack, move the tape head one to the right, and reenter state `pushzero`.

- When in state `next`, the next action of the 2DPDA is found according to the priority list below:
  1. Suppose the current stack symbol is Z_0 (i.e. s_0). Then return REJECT (according to the invariant all binary strings of length n less than or equal $1 \dots 1$ have been rejected).
  2. Suppose the top of the stack is of form $1\ s$. Then pop these two symbols, move the tape head one to the left, and reenter state `next`.
  3. Suppose the top of the stack is of form $0\ s$. Then pop these two symbols, push a 1 followed by $\delta(s, 1)$, and enter state `pushzero` (that the invariant is preserved follows from the fact than being less than or equal $d_1 \dots d_{i-1}\ 0\ 1\ \dots 1$ amounts to being strictly less than $d_1 \dots d_{i-1}\ 1\ 0\ \dots 0$).

If one e.g. considers the DFA which tests whether the input contains an odd number of 1's and an odd number of 0's, the 2DPDA derived by the procedure above can be used as the α_{exp} desired: given N, choose an *odd* n such that $n \geq N$, $n \geq 2$. Now consider a tape τ with $|\tau| = n$ – as it contains $n - 2$ symbols different from $\dagger_L$ and $\dagger_R$, α_{exp} run on such tape will test whether any string of length $n - 2$ is accepted by the DFA. But as $n - 2$ is odd, the DFA will reject all such strings. Hence the 2DPDA enters state `pushzero` at least once for each bit string of length $n - 2$, that is the automaton performs at least 2^{n-2} steps. Hence we infer that

$$T(\alpha_{\text{exp}}, \tau) \geq 4 \cdot 2^{n-2} = 2^n$$

as desired (when the syntactic sugar is eliminated, this inequality will of course still hold!) $\square$

6.3 Speedup possible by using a USM

Given a *deterministic* instance of the USM.
Let $G = \mathit{sim}(\mathit{aut}_i, \mathit{tape}_j, [\,Z_0\,], \sigma_0, 1)$, where aut_i codes automaton α and tape_j codes automaton τ. Assume there is a transition sequence from $\mathsf{SC}(G)$ to a terminal configuration. The $\mathcal{CC}$ thus defined (cf. chapter 5) is termed $\mathcal{CC}(\alpha, \tau)$. On the other hand, if there is an infinite transition sequence from $\mathsf{SC}(G)$ (or the machine gets "stuck") we write $\mathcal{CC}(\alpha, \tau) = \infty$.

Suppose $\mathcal{I}$ is bounded by i_0 (if for instance the PEM is used $i_0 = 2$). According to theorem 5.1.3, then at most a *polynomial speedup* can be achieved[2]. Combining with fact 6.2.1, we have

Fact 6.3.1 Consider a deterministic instance of the USM where $\mathcal{I} \leq i_0$, i_0 a constant. Then there exists a 2DPDA α_{exp}, which terminates on all tapes, with the following property: for all N there exists a $n \geq N$ and a τ with $|\tau| = n$, such that

$$\mathcal{CC}(\alpha_{\mathrm{exp}}, \tau) \geq i_0 \sqrt[i_0]{2^n} = i_0 \cdot 2^{n/i_0}$$

□

On the other hand, in the next section we shall show what amounts to "Cook's theorem":

Theorem 6.3.2 There exists a deterministic instance of the USM and a constant c with the following property: for all automata α and all tapes τ, $\mathcal{CC}(\alpha, \tau) \neq \infty$ iff α terminates when run on τ, in which case

$$\mathcal{CC}(\alpha, \tau) \leq c|\alpha||\tau|$$

□

In particular, there exists a constant c' ($= c \cdot |\alpha_{\mathrm{exp}}|$) such that for all tapes τ we have

$$\mathcal{CC}(\alpha_{\mathrm{exp}}, \tau) \leq c'|\tau|$$

By comparing with fact 6.3.1 we conclude that the machine claimed to exist in theorem 6.3.2 can have no bound on $\mathcal{I}$.

[2][AT92, p. 355] makes the unjustified claim that only a *constant speedup* is possible in such cases.

6.4 A USM implementing Cook's construction

In this section we will prove theorem 6.3.2 by exhibiting a machine, to be called a 2DM, with the desired properties. First some preliminary definitions:

- An expression of form $\mathit{sim}(\alpha, \tau, s, \sigma, \pi)$ with s a *variable* (and α, τ, σ and π constants) and with demand 1 is said to be of type STACK-0 (we know nothing about the stack).
- An expression of form $\mathit{sim}(\alpha, \tau, (\omega_1{::}s), \sigma, \pi)$ with s a variable and with demand 1 is said to be of type STACK-1 (we know the topmost element of the stack).
- An expression of form $\mathit{sim}(\alpha, \tau, (\omega_1{::}(\omega_2{::}s)), \sigma, \pi)$ with s a variable and with demand 1 is said to be of type STACK-2 (we know the two topmost elements of the stack).
- An expression of form $\mathit{sim}(\alpha, \tau, [\, Z_0 \,], \sigma_0, 1)$ with demand 2 is said to be of type INIT (cf. (6.1)).
- An expression of form $\mathit{sim}(\alpha, \tau, [\,], \sigma_0, 1)$ with demand 2 is said to be of type STACK-EMPTY.
- An expression of form ACCEPT or REJECT is said to be of type FINAL.

Defining the 2DM

Let the initial configuration be of form $\mathsf{SC}(G)$, with G of type INIT. The first step of the 2DM will be a `PUSH` step, where an expression of type STACK-1 is pushed (clearly there will be a unique such expression, namely $\mathit{sim}(\alpha, \tau, (Z_0{::}s), \sigma_0, 1)$). The further actions are found according to the priority list below, where we let the current configuration be of form

$$[(G_1, G_1', \_, c_1, n_1, \_) \ldots (G_k, G_k', \_, c_k, n_k, \_)] \tag{6.3}$$

1. *if* G_k' is of form STACK-2 or of form STACK-1 *and* $c_k > 0$ *and* an `UNFOLD` step using a rule at level $i \geq 1$ is possible, *then* do this `UNFOLD` step;

2. *if* G'_k is of form STACK-2 or of form STACK-1 *and* $c_k > 0$ *then* do a `PUSH` step, pushing an expression of type STACK-1. Clearly such an expression will be uniquely defined, if e.g.

   $$G'_k = sim(\alpha, \tau, (\omega_1{::}(\omega_2{::}s)), \sigma, \pi)$$

   then we push $sim(\alpha, \tau, (\omega_1{::}s'), \sigma, \pi)$;

3. *if* it is possible to perform an `UNFOLD` step, using a level 0 rule, then do it;

4. *if* $k > 1$, then perform a `POP` step;

5. *otherwise* halt.

Pushing expressions of type STACK-1 captures Cook's original insight: from knowing the top of the stack one can do a substantial amount of computation, even though the rest of the stack is unknown. The reason for demanding $c_k > 0$ is that otherwise an infinite sequence of `PUSH` steps would be possible.

An invariant of the 2DM

Some preparations are needed for showing that the 2DM satisfies theorem 6.3.2:

Observation 6.4.1 Let the 2DM be in a configuration of form (6.3). Suppose G'_k is of type STACK-1, and suppose $c_k = n_k = 0$. Then, after four steps of the 2DM, it will be in a configuration which is similar, except from the fact that now $c_k = n_k = 4$, and G'_k is either of type STACK-2, of type STACK-1, of type STACK-0 or of type FINAL. □

PROOF: Since $c_k = 0$, neither 1 nor 2 are applicable but 3 is. The 2DM thus performs an `UNFOLD` step using a level 0 rule, and afterwards G'_k will be of form

$$branch(\alpha(\omega_1, \sigma, \tau(\pi)), \alpha, \tau, \omega_1, s, \pi)$$

Next reduce the redex $\tau(\pi)$; then reduce the redex $\alpha(\ldots)$ and finally reduce the redex $branch(\ldots)$ – all of those redices have demand 1. □
Conceptually, we will consider the four steps mentioned in observation 6.4.1 as one single step. By doing so, we are able to formulate

Lemma 6.4.2 After the first step, the 2DM obeys the following invariant (let the current configuration be of form (6.3)):

1. For $j \in \{2 \ldots k\}$, G_j is of type STACK-1.
2. For $j \in \{2 \ldots k-1\}$, $n_j > 0$.
3. For $j \in \{2 \ldots k-1\}$, *either* G'_j is of type STACK-2 and $c_j = 4$ *or* G'_j is of type STACK-1 and $4 \leq c_j \leq 5$.
4. If $(\_ : G \Rightarrow^{c}_{Nn} G') \in \mathcal{R}_i$, $i \geq 1$, then G is of type STACK-1, G' is of type STACK-0 or of type FINAL, $n > 0$ and $4 \leq c \leq 6$.
5. If $k > 1$, then *either*
   - $G'_k = G_k$, $c_k = n_k = 0$ *or*
   - G'_k is of type STACK-2, $c_k = 4$, $n_k > 0$ *or*
   - G'_k is of type STACK-1, $4 \leq c_k \leq 5$, $n_k > 0$ *or*
   - G'_k is of type STACK-0, $4 \leq c_k \leq 6$, $n_k > 0$ *or*
   - G'_k is of type FINAL, $4 \leq c_k \leq 6$, $n_k > 0$.
6. G_1 is of type INIT, and *either* $k \geq 2$ and $G'_1 = G_1$ and $c_1 = n_1 = 0$ *or* $k = 1$ and $c_1 = 1$, $n_1 > 0$ and G'_1 is of type FINAL or of type STACK-EMPTY.

□

PROOF: It is immediate that the invariant holds after the first step of the 2DM; let us now check that part 1-6 is preserved by any subsequent step of the 2DM:

1. Suppose an `UNFOLD` step is performed, using a rule in $\mathcal{R}_i$ with $i \geq 1$. The only interesting point is to check that part 5 still holds. As 5 holds *before* the step, we from $c_k > 0$ can infer that $n_k > 0$ and that either G'_k is of type STACK-2 with $c_k = 4$ or G'_k is of type STACK-1 with $4 \leq c_k \leq 5$. Due to the form of the rules[3] (part 4), *after* the step it in the former case holds that G'_k is of type STACK-1 or of type FINAL with $c_k = 5$; and in the latter case it holds that G'_k is of type STACK-0 or of type FINAL with $5 \leq c_k \leq 6$.

[3]If there for instance is a rule, representing a reduction from $sim(\alpha, \tau, (\omega::s), \sigma, \pi)$ to $sim(\alpha, \tau, s, \sigma_1, \pi_1)$, then this rule can be used to generate a reduction from $sim(\alpha, \tau, (\omega::(\omega_1::s)), \sigma, \pi)$ to $sim(\alpha, \tau, (\omega_1::s), \sigma_1, \pi_1)$.

2. Suppose a `PUSH` step is performed. Since part 5 holds before the step, we from $c_k > 0$ can infer that $n_k > 0$ and that either G'_k is of type $\mathsf{STACK\text{-}2}$, $c_k = 4$ or G'_k is of type $\mathsf{STACK\text{-}1}$, $4 \leq c_k \leq 5$. Now it is easy to see that the invariant is preserved.

3. Suppose a (sequence of, cf. observation 6.4.1) `UNFOLD` step is made, using level 0 rules. The only interesting point is to check that part 5 still holds. G'_k cannot be of type FINAL or of type $\mathsf{STACK\text{-}0}$, as such expressions cannot be reduced further. Neither can G'_k be of type $\mathsf{STACK\text{-}2}$ or $\mathsf{STACK\text{-}1}$ with $c_k > 0$, as then an `UNFOLD` step using a rule at a level ≥ 1 or a `PUSH` step would have been made. As part 5 holds before the step, we hence infer that $G'_k = G_k$ and $c_k = n_k = 0$. From part 1 we see that G'_k is of type $\mathsf{STACK\text{-}1}$; now observation 6.4.1 gives the claim.

4. Suppose a `POP` step is made. Since part 5 holds before the step, and no other steps were possible, we conclude that $n_k > 0$, $4 \leq c_k \leq 6$ and that G'_k is of type $\mathsf{STACK\text{-}0}$ or of type FINAL. It is now immediate that part 1-4 still hold. If $k > 2$ before the step we must show that part 5 still holds – this is done by observing that if $k > 2$ then by part 3 either G'_{k-1} is of type $\mathsf{STACK\text{-}2}$ with $c_{k-1} = 4$ or G'_{k-1} is of type $\mathsf{STACK\text{-}1}$ with $4 \leq c_{k-1} \leq 5$.

   Finally, if $k = 2$ before the step we must show that part 6 still holds. But this is obvious: if G'_2 is of type FINAL also G'_1 will become of type FINAL; and if G'_2 is of type $\mathsf{STACK\text{-}0}$ G'_1 (as G_1 is of type INIT) will become of type $\mathsf{STACK\text{-}EMPTY}$.

$\square$

Partial correctness

Now we can prove that the 2DM is partially correct:

Theorem 6.4.3 Let $G = \mathit{sim}(\alpha, \tau, [\,Z_0\,], \sigma_0, 1)$. Suppose the 2DM makes the transition sequence

$$\mathsf{SC}(G) \;\rhd^* [(G, G', \_, c, n, i)]$$

and then halts. Then G' is of type FINAL, and the automaton α run on the tape τ returns the answer indicated by G'. $\square$

PROOF: According to lemma 6.4.2 part 6, G' is of type FINAL or of type STACK-EMPTY. Theorem 5.1.3 now says (still with our implicit coding terms/graphs) that $1 \vdash^* \_ : G \Rightarrow_{\bar{N}\_} G'$. But due to requirement 2 on page 147, we infer that G' cannot be of type STACK-EMPTY. This concludes the proof. □

Total correctness

Also total correctness holds:

Theorem 6.4.4 Let $G = sim(\alpha, \tau, [\,Z_0\,], \sigma_0, 1)$, and suppose the 2DM when started in configuration $\mathsf{SC}(G)$ loops. Then the automaton α loops when run on tape τ. □

PROOF: We want to apply theorem 5.1.7; to do so we have to show that any execution sequence of the 2DM is progressing. But this follows from lemma 6.4.2: by part 4 all `POP` steps are progressing and all `UNFOLD` steps using rules at level ≥ 1 are progressing, and it is easy to see that also all `UNFOLD` steps using level 0 rules are progressing. And by the preconditions for a `PUSH` step, there cannot be an infinite sequence of such steps. □

Complexity of the 2DM

Suppose the 2DM terminates. Then, by lemma 6.4.2 part 6, we have

$$\mathcal{CC} = 1 + \mathcal{TT}_i, i = \mathcal{I} - 1$$

Let N be the total number of rules at level ≥ 1. By fact 5.1.1, $N \geq \mathcal{I} - 1$. Also $N \geq 1$, as the 2DM starts by doing a `PUSH` step. Exploiting lemma 6.4.2 part 4, which states that the maximum cost of a rule is 6, we infer that

$$\mathcal{CC} \leq 1 + \mathcal{I} - 1 + 6N \leq N + N + 6N \leq 8N \tag{6.4}$$

Let us put some bound on N: suppose $(\_ : G_1 \Rightarrow^{c_1}_{Nn_1} G_1') \in \mathcal{R}_{i_1}$ and $(\_ : G_2 \Rightarrow^{c_2}_{Nn_2} G_2') \in \mathcal{R}_{i_2}$, $i_1, i_2 \geq 1$. By lemma 6.4.2 part 4, G_1 and G_2 both are of type STACK-1, ie. they are of form

$$G_1 = sim(\alpha, \tau, (\omega_1{::}s), \sigma_1, \pi_1), G_2 = sim(\alpha, \tau, (\omega_2{::}s), \sigma_2, \pi_2)$$

(it is easy to see that the α's and τ's are identical). We want to show that there is at most one rule for each triple (ω, σ, π): assume for the sake of getting a contradiction that an expression $\mathit{sim}(\alpha, \tau, (\omega{::}s), \sigma, \pi)$ is pushed twice. Since a `PUSH` step is not performed if a corresponding rule already exists, the second push is done before the stack of the 2DM has returned to the height it had when performing the first push. But then it it easy to see that the 2DM will loop.

As there by definition of $|\alpha|$ and $|\tau|$ exist $|\alpha|{\cdot}|\tau|$ triples of form (ω, σ, π), we by (6.4) conclude that

$$\mathcal{CC} \leq 8|\alpha||\tau|$$

The above, together with theorem 6.4.3 and theorem 6.4.4, constitutes a proof of theorem 6.3.2 (with $c = 8$).

Further remarks

- Unlike many other applications of an ultimate sharing machine, $\mathcal{CC}$ in this case certainly is a "realistic" cost measure: the 2DM is completely deterministic; the rules can be conveniently stored in a three-dimensional array such that they can be retrieved in constant time; and the "size" of the rules is bounded – cf. the "pseudo-speedup" encountered in the *Towers of Hanoi* example from section 4.7.4.

- From the above we see that the halting problem is solvable for 2DPDAs: if the 2DM does not stop[4] before $\mathcal{CC}$ exceeds $8|\alpha||\tau|$, we (from theorem 6.3.2 where we know $c = 8$ can be used) can deduce that α loops on τ.

  Another way to detect loops would be to keep track of whether the 2DM twice performs a `PUSH` step pushing the same expression.

6.5 Previous work

The original technique for linear time simulation of a 2DPDA, used in [Coo71] and restated as [AHU74, algorithm 9.4], is a *bottom-up* approach (cf. page 20). We will now hint at how to translate this method into our

[4]As no infinite sequence of `PUSH` steps is possible, fact 5.1.2 tells us that the 2DM loops iff $\mathcal{CC}$ can become arbitrarily big.

framework: given an expression e of type $\mathsf{STACK\text{-}1}$, the *terminator* of e is the unique (but possibly non-existent) expression e' of type $\mathsf{STACK\text{-}1}$ such that there is a level 1 transition from e to e', and such that e' in one step reduces to an expression of type $\mathsf{STACK\text{-}0}$ or of type FINAL.

The purpose of the algorithm is to build up the level i rules, each rule associating an expression with its terminator – initially, $\mathcal{R}_1$ will contain those e which are their own terminators, i.e. those e which in one step reduce to an expression of type $\mathsf{STACK\text{-}0}$ or of type FINAL. The set of rules is now extended during the main loop, where one for instance exploits that if e in one step reduces to e', and the terminator of e' is e'', then also e has e'' as terminator. When all (existing) terminators have been computed, the "answer" of the simulation can be looked up immediately.

The above algorithm typically generates a lot of rules which are never needed for computing the final answer, cf. the discussion page 20. Motivated by this observation [Jon77] presents what amounts to a top-down version of the algorithm, which gives rise to essentially the same "flow-of-control" as the 2DM.

In [Wat80] it is shown how something similar to [AHU74, algorithm 9.4] can be derived by means of general tabulation techniques.

Even though Cook's insight gives a linear time algorithm for many problems where such an algorithm is not in any way obvious, the constant factor may be prohibitively large. [ANTJ89] aims at decreasing this constant factor by *partially evaluating* a modified version of the top-down simulator from [Jon77] wrt. known automata.

Chapter 7

Deriving efficient substring matchers by partial evaluation

In this chapter we show that the well-known Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) algorithms for substring matching can be seen as instances of a *common* substring matching algorithm, *parametrized* wrt. search strategy. The purpose of this is twofold:

1. to formalize the intuition that KMP and BM are "dual";
2. to show how – by a sequence of small steps, none of which requires much ingenuity – it is possible to (re)invent ingenious constructions, thus making us learn something about how to devise efficient algorithms. One may argue that this is of limited practical use, as it seems unlikely that one by means of general constructions can come up with something undiscovered by the advanced art of algorithmics – on the other hand, recall from p. 145 that Knuth actually got the inspiration to the KMP algorithm from Cook's general 2DPDA simulation algorithm.

7.1 Introduction

Substring matching (in our formulation) amounts to determining the position of the first occurrence of a string p (called the pattern string) in another string s (called the subject string). Wlog. we can assume that p occurs in s (e.g. by appending p to the end of s). We refer to p and s as arrays, so e.g. $p[i]$ ($i \in \{0 \ldots |p| - 1\}$) denotes the i'th element of p (this is termed a *reference* to p). A naive algorithm to solve the problem is given below[1]:

$$subs0(p,s) = loop(p,0,s,0)$$
$$loop(p,i,s,j) = j, \texttt{ if } i = |\ p\ |$$
$$loop(p,i,s,j) = loop(p,i+1,s,j), \texttt{ if } p[i] = s[j+i]$$
$$loop(p,i,s,j) = loop(p,0,s,j+1), \texttt{ if } p[i] \neq s[j+i]$$

The worst case complexity of this algorithm is $\Theta(|p| \cdot |s|)$ (to see this, let e.g. s as well as p consist of A's only except for the last element).

Rather than using the above *one-stage* algorithm, we shall be interested in *two-stage* algorithms[2]. That is, given p and s

1. (the *preprocessing* phase:) a program M_p is constructed such that M_p run on (any) s gives the same result as *subs0* run on p and s (but is hopefully quicker!)
2. M_p is run on s.

We will demand that M_p makes no references to p; the complexity of M_p will be measured as the number of references to s.

There are several ways to come up with such M_p:

- to use the KMP method (with some abuse of notation we identify the KMP algorithm with the behavior of M_p thus obtained);
- to use (some variant of) the BM method (we identify the BM algorithm with the behavior of M_p thus obtained);
- to apply the technique of partial evaluation to (a modified version of) *subs0.*

In the rest of this chapter, we proceed as follows: in section 7.2 we sketch how the KMP method works and in section 7.3 we sketch how the BM method(s) work, by means of examples. In section 7.4 we rewrite *subs0* into a program *subs2*, parametrized wrt. search strategy. In section 7.5 we discuss various kinds of search strategies; by means of examples we will show that by doing PE on *subs2* wrt. certain "natural" search strategies (and wrt. fixed p) one achieves target programs implementing the KMP algorithm/BM algorithms. Finally, section 7.6 compares with related work.

Some remarks concerning our treatment:

[1]It is easy to see how it can be coded as level 0 rules.

[2]For a survey of pattern matching algorithms, see [Aho90].

- we will *not* formally prove that the target programs indeed implement the KMP/BM algorithms, as this would require a rather heavy machinery;
- no attention will be paid to the complexity of the preprocessing phase;
- as it is a rather common exercise to achieve the KMP algorithm by PE (see e.g. [Smi91] for a survey), we primarily focus upon the BM algorithm.

7.2 The KMP method

This is the method described in [KMP77]. As made explicit in [AHU74, Algorithm 9.3], it amounts to constructing from p a DFA which given s as input enters its accepting state exactly at the end of the first occurrence of p in s. Thus at most $|s|$ references to s are made.

The DFA has a state for each position in p; the behavior is best clarified by an example: let $p =$AABAAA, then there will exist states $S_0 \ldots S_6$ with S_0 the initial state and S_6 the accepting state. The interpretation of the DFA being in state S_6 is that the previous 6 symbols of s have been AABAAA, the interpretation of the DFA being in state S_5 is that the previous 5 symbols of s have been AABAA,..., the interpretation of the DFA being in state S_3 is that the previous 3 symbols of s have been AAB, ..., the interpretation of the DFA being in state S_1 is that the previous symbol has been A (and the preceding symbols have been such that the criteria for being in a higher state are not satisfied); and the interpretation of the DFA being in state S_0 is that the criteria for being in a higher state are not satisfied.

Let us now discuss the behavior of the DFA when in state S_5 (thus the last 5 symbols are AABAA):

- If the next symbol is an A, the accepting state S_6 can be entered.
- If the next symbol is a B, the situation is now that the previous 6 symbols have been AABAAB. Thus we are not allowed to enter S_6, S_5 or S_4; but S_3 can be entered.
- If the next symbol is neither A nor B, none of the $S_1 \ldots S_6$ fit – hence S_0 is entered.

7.3 The BM methods

Originally devised by Boyer and Moore [BM77], numerous variants have emerged – some of which are described in e.g. [KMP77] and [PS88]. The basic idea is to do the matching *backwards* wrt. p: with $p =$ TREE (which will be our working example), the initial configuration will be

```
...x.........
   ^
TREE
```

If x is an E, we move the pointer one to the left and attempt to match the first E of TREE, etc. The various algorithms differ in the way they treat *mismatches*; the original approach [BM77] is to maintain two tables, δ_1 and δ_2. δ_1 has an entry for each symbol in the alphabet (the size of which will be denoted Σ) which contains the *rightmost* occurrence (if any) of the symbol in p. δ_2 has an entry for each position in p containing information about the rightmost occurrence of the corresponding suffix[3] (if any) elsewhere in p (δ_2 is thus roughly of size $|p|$).

We shall consider the following 4 variants of the BM method, ordered after increasing degree of sophistication (the naming used is in no way conventional):

1. The *naive Boyer-Moore*, to be called BMna. This is a simplification of the one given in [BM77], exploiting δ_1 only.

2. The *original Boyer-Moore*, to be called BMor. This is the one presented in
[BM77], i.e. δ_1 as well as δ_2 is exploited.

3. The *standard Boyer-Moore*, to be called BMst. This is the algorithm hinted at
[BM77, p. 771, below] and the one derived in [PS88]; it works by "combining" δ_1 and δ_2 into a single two-dimensional table.

4. The *optimal Boyer-Moore*, to be called BMop. This is the algorithm suggested by Knuth
[p. 346][KMP77]; here no information is discarded (hence a larger data structure than δ_1/δ_2 is needed).

[3]Moreover, this occurrence must be preceded by a different symbol.

We will illustrate the behavior of these algorithms by giving examples where one algorithm behaves in a suboptimal fashion, whereas the “next one” in the ordering behaves optimally.

7.3.1 BMna vs. BMor

Let the situation be

```
.TEE...
 ^
TREE
```

i.e. the two E's have been found to match, but then a mismatch is detected. As the symbol currently scanned in the subject string is a T, and as δ_1 records that T occurs in position 0 in p (but not elsewhere), BMna will shift the pattern string one position to the right yielding the configuration

```
.TEE...
    ^
 TREE
```

However, the fact that the symbols to the right of the mismatch in the subject string are known to be EE actually justifies shifting the pattern string *four* positions to the right, as done by BMor.

7.3.2 BMor vs. BMst

Let the situation be

```
..TE..
  ^
TREE
```

i.e. the last E has been successfully matched.

1. From δ_1 we can infer that it is sound to shift the pattern string two to the right, as T (the symbol scanned in the subject string) occurs in position 0 in p (but not elsewhere).

2. From δ_2, which only exploits that the next symbol in the subject string is an E and the current symbol is not an E (but does *not* exploit that the current symbol is a T), we can infer nothing more than it is sound to shift the pattern string one to the right.

Hence BMor will move the pattern string $\max(2,1) = 2$ positions to the right. This is very conservative, since from knowing that the symbol scanned is a T and the symbol to its right is an E it is possible to shift the pattern string four positions to the right, as done by BMst.

7.3.3 BMst vs. BMop

Let the situation be

```
MOORRE....
   ^
TREE
```

By exploiting δ_1 (alone) we can shift the pattern string two to the right, and after E has been successfully matched the situation is

```
MOORRE....
    ^
  TREE
```

Now BMst is able to move the pattern string one to the right only, since it knows that then R and the first E will match. BMop does better than that, since it remembers that the fourth symbol of the subject string is an R – hence it is possible to shift the pattern string four to the right.

[BM77, p. 764] illustrates the behavior of BMor by means of an example, the pattern string being AT-THAT and the subject string s being

```
WHICH-FINALLY-HALTS.--AT-THAT-POINT
```

BMor on that example makes 14 references to s. It will be a useful exercise to verify that BMop makes 11 references only, BMst makes 12 and BMna 16.

7.3.4 Discussion

Several remarks can be made about the merits of the various Boyer-Moore algorithms:

- It will not always be the case that a “more powerful” method is faster, since it may be useful to make a small shift initially in order to make larger shifts later on. As an example of this, consider (as in [KMP77, p. 343]) the following situation:

```
ABABABABABABABABABABABA...
        ^
AAAAAAACB
```

δ_1 enables a double shift, whereas δ_2 enables a single shift only. However, after this single shift the situation will soon be

```
ABABABABABABABABABABABA...
        ^
 AAAAAAACB
```

and now δ_2 enables a maximal shift.

- In [KMP77, p. 343] it is shown that BMor makes at most $6|s|$ references to s (the constant 6 probably being much too large). The linearity is solely [BM77, p. 767] due to the use[4] of δ_2. If only δ_1 is used (i.e. BMna is considered), $\Theta(|p||s|)$ references to s may be made (if e.g. p is of form ABBBBBB... and a large initial segment of s contains B's only). However, BMna will suffice for most practical purposes, especially if the alphabet is large.

- In [BM77] a theoretical analysis of BMor is given, and the theoretical performance is compared with the actual performance. For large alphabets the theoretical analysis is completely accurate, but for the binary alphabet actual performance is significantly worse than the one predicted by the theoretical model. As explained by the authors [BM77, p. 770], this is because the model does not account for the fact that some matches are guaranteed to occur. For instance, when we from the configuration

```
......BAA...
      ^
AAABAAAAA
```

shift the pattern string three to the right the possibility that a mismatch occurs in position 3,4 or 5 is zero – as AAABAAAAA has been aligned in such a way that its substring BAA matches the subject string – whereas it in the simplified theoretical model is

[4]As pointed out in [BM77, p. 771] it is necessary in order to get linearity that δ_2 also records the *preceding* symbol, cf. footnote 3.

non-zero. For small alphabets, the length of such aligned substrings can be considerable.

- BMst is typically only slightly faster than BMor, but uses somewhat more space: for each position in the pattern string, a table with potentially Σ entries has to be created. However, in practice the space required will be $O(|p|)$, since only a few of these entries will differ from the default action, allowing a compact representation of the tables.

- The merits of BMop (versus BMst) are briefly discussed in [KMP77, p. 346]. It is trivial that BMop makes at most $|s|$ references to s (it never looks at the same position twice, as no information is forgotten). However, only for small alphabets we may expect BMop to be significantly faster than BMst. On the other hand, as the alphabet grows smaller the table of BMop may grow really large, even though it is doubtful whether $2^{|p|}$ really is the tightest upper bound (for a further discussion of this issue, see [Cho90]). If the alphabet is large, we may expect BMop to use space $O(|p|^2)$.

7.4 Rewriting *subs0*

We now rewrite the naive substring matcher *subs0* from section 7.1 into a substring matcher *subs1* which exploits previously gained information and further into a substring matcher *subs2* which is parametrized wrt. search strategy. We hope the reader will agree that the development is quite “natural” and does not require much ingenuity.

Making previously gained information explicit

The reason why *subs0* is potentially inefficient is that the same element of s may be referenced several times, since the content is “forgotten” immediately after. To repair on this we introduce the notion of an *information array*, the elements of which are either of form $\textsf{Pos-info}(x)$ (with x belonging to the alphabet); of form $\textsf{Neg-info}(X)$ (with X being a subset of the alphabet) or of form $\textsf{No-info}$.

We say that an information array *inf* is *sound* wrt. s and j if it for all i in the domain of *inf* holds that

- if *inf*[i] = Pos-info(x), then $s[j+i] = x$;
- if *inf*[i] = Neg-info(X), then $s[j+i] \notin X$.

If e.g. *inf*[0] = Pos-info(A), *inf*[1] = No-info and *inf*[2] = Neg-info($\{$A, B$\}$), we shall write *inf* = [A, $\emptyset$, $\{$A, B$\}$].

We next introduce some operations on information arrays:

- Empty(k) returns an information array of size k (indexed from 0 to $k-1$), where all elements are No-info.
- Shift(*inf*), *inf* having size k, returns an information array *inf'* of size k where *inf'*[i] = *inf*[$i+1$] for $i < k-1$, *inf'*[$k-1$] = No-info. Hence, if *inf* is sound wrt. s and j then Shift(*inf*) is sound wrt. s and $j+1$.
- The predicate Certain?(*inf*, i, x) returns true iff *inf*[i] = Pos-info(x). Thus, if Certain?(*inf*, i, x) holds and *inf* is sound wrt. s and j we can infer that $s[j+i] = x$.
- The predicate Impossible?(*inf*, i, x) returns true iff *either* *inf*[i] = Pos-info(x'), $x \neq x'$ *or* *inf*[i] = Neg-info(X), $x \in X$. Thus, if Impossible?(*inf*, i, x) holds and *inf* is sound wrt. s and j we can infer that $s[j+i] \neq x$.
- Addpos(*inf*, i, x) returns *inf'*, where *inf'*[i'] = *inf*[i'] for $i' \neq i$, and where *inf'*[i] = Pos-info(x).
- Addneg(*inf*, i, x) returns *inf'*, where *inf'*[i'] = *inf*[i'] for $i' \neq i$, and where *inf'*[i] = Neg-info($\{x\}$) if *inf*[i] = No-info; *inf'*[i] = Neg-info($x \cup X$) if *inf*[i] = Neg-info(X).

We are now in position to rewrite *subs0* into *subs1* – it should be straightforward to verify that *subs1* is "correct", in particular that *inf* always is sound wrt. s and j:

```
subs1(p,s)         = loop(p,0,s,0,Empty(| p |))
loop(p,i,s,j,inf)  = j, if  i = | p |
loop(p,i,s,j,inf)  = loop(p,i+1,s,j,inf), if  Certain?(inf,i,p[i])
loop(p,i,s,j,inf)  = loop(p,0,s,j+1,Shift(inf)), if  Impossible?(inf,i,p[i])
loop(p,i,s,j,inf)  = loop(p,i+1,s,j,Addpos(inf,i,p[i])), if  p[i] = s[j+i]
loop(p,i,s,j,inf)  = loop(p,0,s,j+1,Shift(Addneg(inf,i,p[i]))), if  p[i] ≠ s[j+i]
```

Of course, *subs1* is no more efficient than *subs0* – but the point is that the operations Certain? and Impossible? can be carried out at PE time.

Generalizing the search strategy

For each value of j in the program above, the variable i sequentially assumes the value $0, 1, 2, \ldots$ until either a mismatch is detected or until the value $|p|$ is reached. Alternative search strategies might be useful; therefore we now introduce an extra parameter *stra* representing the search strategy in question. Given an information array *inf* of size k, *stra* must return a sequence where each element *either* is an integer belonging to the interval $\{0 \ldots k-1\}$ *or* is a special element FORGET. Of course, we must require each $i \in \{0 \ldots k-1\}$ to occur at least once in the sequence. Some noteworthy points (to be explained further in section 7.5.1 and in section 7.5.5):

- The search strategy depends on the current information array. Intuitively, it will be useful to search the "known" elements first as these tests may be eliminated at PE time.
- The element FORGET tells us to forget the current information. This may be useful in order to prevent the information array from growing too large…
- An element of $\{0 \ldots k-1\}$ may occur more than once in the sequence – this may seem stupid, but correctness is of course not affected.

Now it is straight-forward to write *subs2*:

```
subs2(p,s,stra)  = loop(p,stra(initinf),s,0,initinf,stra)
                   where  initinf = Empty(| p |)

loop(p,[ ],s,j,inf,stra)  = j

loop(p,[FORGET|iseq],s,j,inf,stra)  = loop(p,iseq,s,j,Empty(| p |),stra)

loop(p,[i|iseq],s,j,inf,stra)  = loop(p,iseq,s,j,inf,stra),
                                 if  Certain?(inf,i,p[i])

loop(p,[i|_],s,j,inf,stra)  = loop(p,stra(newinf),s,j+1,newinf,stra),
                              if  Impossible?(inf,i,p[i])
                              where  newinf = Shift(inf)

loop(p,[i|iseq],s,j,inf,stra)  = loop(p,iseq,s,j,Addpos(inf,i,p[i]),stra),
                                   if   $p[i] = s[j{+}i]$
$loop(p,[i|\_],s,j,inf,stra)$  $=$ $loop(p,stra(newinf),s,j{+}1,newinf,stra)$,
                                 if   $p[i] \neq s[j{+}i]$
                                 where   $newinf = \mathsf{Shift}(\mathsf{Addneg}(inf,i,p[i]))$
```

7.5 PE wrt. various search strategies

In this section we investigate various search strategies, and sketch the result of doing PE on *subs2* wrt. these strategies (and wrt. fixed p). We will proceed as follows: in section 7.5.1 we argue that there are two "natural" choices for an "optimal" search strategy, to be denoted KMP and BMop. In section 7.5.2 and section 7.5.3 we will give examples of the behavior of KMP and BMop, hopefully convincing the reader that KMP (after PE) gives rise to the KMP algorithm and that BMop (after PE) gives rise the BMop algorithm. In section 7.5.4 we present a (not quite natural) search strategy BMst which gives rise to the BMst algorithm. Section 7.5.5 contains a brief discussion of the approach.

7.5.1 "Natural" search strategies

First some notation:

- Pos(*inf*) denotes the *ascending* sequence of indices i such that *inf*$[i]$ is of form Pos-info(x).
- Neg(*inf*) denotes the *ascending* sequence of indices i such that *inf*$[i]$ is of form Neg-info(X).
- Nothing(*inf*) denotes the *ascending* sequence of indices i such that *inf*$[i]$ = No-info.
- All(*inf*) denotes the sequence $0, 1, 2 \ldots k-1$, where k is the size of *inf*.

So if e.g. *inf* $= [\,\text{A}, \emptyset, \text{B}, \{\text{AB}\}, \emptyset\,]$ we have Pos(*inf*) = [0,2], Neg(*inf*) = [3], Nothing(*inf*) = [1,4] and All(*inf*) = [0,1,2,3,4].

Let us now embark on deducing what constitutes an "optimal" and "natural" search strategy: it seems clear that it should start by examining the positions in Pos, since these positions can be completely tested for match without examination of s. Also it seems clear that the positions in

Neg should be tested before the positions in $\mathsf{Nothing}$, since we then may be able to discover a mismatch without examination of s. We have thus argued that an “optimal” search strategy takes the form

$$p_1(\mathsf{Pos}) +\!\!+\ p_2(\mathsf{Neg}) +\!\!+\ p_3(\mathsf{Nothing})$$

where p_1, p_2 and p_3 are permutations. It is easily seen that as long as Neg is processed before the elements in $\mathsf{Nothing}$, there cannot be more than one position in Neg. With the strategy above, Neg is thus assured to be either empty or a singleton. The performance of the target program will not be affected by the choice of p_1 (the choice might affect the complexity of the preprocessing phase).

So the only interesting issue is the choice of p_3. Two possibilities can be considered “natural”: the identity permutation and the reversing permutation. The corresponding search strategies will be christened KMP and BMop:

$$\begin{aligned} \mathsf{KMP} &= \mathsf{Pos} +\!\!+\ \mathsf{Neg} +\!\!+\ \mathsf{Nothing}, \\ \mathsf{BMop} &= \mathsf{Pos}^R +\!\!+\ \mathsf{Neg} +\!\!+\ \mathsf{Nothing}^R \end{aligned}$$

It is easily seen that KMP can be simply reexpressed as

$$\mathsf{KMP} = \mathsf{All}$$

i.e. *subs2* equipped with search strategy KMP is “isomorphic” to *subs1*.

7.5.2 Obtaining KMP via KMP

We now, by means of an example, sketch the algorithm resulting from doing PE on *subs2* wrt. fixed p and the search strategy KMP.

It is helpful, for $i \in \{0 \ldots |p| - 1\}$, to introduce $S_i(s,j)$ as an abbreviation for

$$loop(p, [\,i, i+1, \ldots, |p|-1\,], s, j, [\,p[0], \ldots, p[i-1], \emptyset, \ldots, \emptyset\,], \mathsf{KMP})$$

Now all functions in the target program will be of form

$S_i(s,j) \rightarrow \varepsilon$, `if` $s[j+i] = \ldots$

where *either* $\varepsilon = j$ (denoting accept) *or* ε is of the form $S_{i'}(s,j')$, with $i'+j' = i+j+1$ – i.e. the “subject string pointer” has been advanced by one.

To make the discussion concrete, let $p =$ AABAAA and let us find the code for $S_5(s,j)$ – cf. section 7.2.

First assume $s[j{+}5] = \text{A}$. Then we have the derivation

$$\begin{array}{ll} & S_5(s,j) \\ = & loop(\text{AABAAA}, [\,5\,], s, j, [\,\text{A}, \text{A}, \text{B}, \text{A}, \text{A}, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\;], s, j, [\,\text{A}, \text{A}, \text{B}, \text{A}, \text{A}, \text{A}\,], \mathsf{KMP}) \\ \rightarrow & j \end{array}$$

enabling us to store the rule

$$S_5(s,j) \rightarrow j, \;\; \texttt{if} \;\; s[j{+}5] = \text{A}. \tag{7.1}$$

Next assume $s[j{+}5] \neq \text{A}$. Then we have the derivation

$$\begin{array}{ll} & S_5(s,j) \\ = & loop(\text{AABAAA}, [\,5\,], s, j, [\,\text{A}, \text{A}, \text{B}, \text{A}, \text{A}, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,0,1,2,3,4,5\,], s, j{+}1, [\,\text{A}, \text{B}, \text{A}, \text{A}, \{\text{A}\}, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,1,2,3,4,5\,], s, j{+}1, [\,\text{A}, \text{B}, \text{A}, \text{A}, \{\text{A}\}, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,0,1,2,3,4,5\,], s, j{+}2, [\,\text{B}, \text{A}, \text{A}, \{\text{A}\}, \emptyset, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,0,1,2,3,4,5\,], s, j{+}3, [\,\text{A}, \text{A}, \{\text{A}\}, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,1,2,3,4,5\,], s, j{+}3, [\,\text{A}, \text{A}, \{\text{A}\}, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,2,3,4,5\,], s, j{+}3, [\,\text{A}, \text{A}, \{\text{A}\}, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \end{array}$$

which can be continued in two ways, depending on whether $s[j{+}5] = \text{B}$ or not. In the former case the derivation

$$\begin{array}{ll} & loop(\text{AABAAA}, [\,2,3,4,5\,], s, j{+}3, [\,\text{A}, \text{A}, \{\text{A}\}, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,3,4,5\,], s, j{+}3, [\,\text{A}, \text{A}, \text{B}, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \\ = & S_3(s, j{+}3) \end{array}$$

enables us to store the rule

$$S_5(s,j) \rightarrow S_3(s, j{+}3), \;\; \texttt{if} \;\; s[j{+}5] = \text{B}. \tag{7.2}$$

In the latter case the derivation

$$\begin{array}{ll} & loop(\text{AABAAA}, [\,2,3,4,5\,], s, j{+}3, [\,\text{A}, \text{A}, \{\text{A}\}, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,0,1,2,3,4,5\,], s, j{+}4, [\,\text{A}, \{\text{A}, \text{B}\}, \emptyset, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,1,2,3,4,5\,], s, j{+}4, [\,\text{A}, \{\text{A}, \text{B}\}, \emptyset, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,0,1,2,3,4,5\,], s, j{+}5, [\,\{\text{A}, \text{B}\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \\ \rightarrow & loop(\text{AABAAA}, [\,0,1,2,3,4,5\,], s, j{+}6, [\,\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\,], \mathsf{KMP}) \\ = & S_0(s, j{+}6) \end{array}$$

enables us to store the rule

$$S_5(s,j) \rightarrow S_0(s,j{+}6), \ \texttt{if} \ \ s[j{+}5] \notin \{\textsc{a}, \textsc{b}\}. \qquad (7.3)$$

Referring back to section 7.2, we see that the actions of the DFA when in state S_5 are closely mirrored by the rules (7.1), (7.2) and (7.3) just derived.

We may be tempted to conclude that the KMP strategy gives rise to the KMP algorithm ...

7.5.3 Obtaining BMop via BMop

We now investigate the algorithm resulting from doing PE on *subs2* wrt. $p = \textsc{tree}$ and the search strategy BMop; our aim will be to show that it exhibits the same behavior in the example from section 7.3.3 as BMop. This will be the case provided the target program contains the following rules, with S the "main function":

$S(s,j) \rightarrow S_1(s,j{+}2)$, `if` $s[j{+}3] = \textsc{r}$

$S_1(s,j) \rightarrow S_{1,3}(s,j)$, `if` $s[j{+}3] = \textsc{e}$

$S_{1,3}(s,j) \rightarrow S(s,j{+}4)$, `if` $s[j{+}2] \neq \textsc{e}$

In order to show this, first let S, S_1 and $S_{1,3}$ be abbreviations as defined below:

$S(s,j) \ = loop(\textsc{tree}, [\,3,2,1,0\,], s,j, \ [\,\emptyset, \emptyset, \emptyset, \emptyset\,], \mathsf{BMop})$

$S_1(s,j) \ = loop(\textsc{tree}, [\,3,2,0\,], s,j, \ [\,\emptyset, \textsc{r}, \emptyset, \emptyset\,], \mathsf{BMop})$

$S_{1,3}(s,j) \ = loop(\textsc{tree}, [\,2,0\,], s,j, \ [\,\emptyset, \textsc{r}, \emptyset, \textsc{e}\,], \mathsf{BMop})$

Then the claim follows from the calculations below:

1. if s[j+3] = R,

$$\begin{array}{ll} & S(s,j) \\ = & loop(\textsc{tree}, [\,3,2,1,0\,], s, j, [\,\emptyset, \emptyset, \emptyset, \emptyset\,], \mathsf{BMop}) \\ \rightarrow & loop(\textsc{tree}, [\,2,3,1,0\,], s, j{+}1, [\,\emptyset, \emptyset, \{\textsc{e}\}, \emptyset\,], \mathsf{BMop}) \\ \rightarrow & loop(\textsc{tree}, [\,1,3,2,0\,], s, j{+}2, [\,\emptyset, \{\textsc{e}\}, \emptyset, \emptyset\,], \mathsf{BMop}) \\ \rightarrow & loop(\textsc{tree}, [\,3,2,0\,], s, j{+}2, [\,\emptyset, \textsc{r}, \emptyset, \emptyset\,], \mathsf{BMop}) \\ = & S_1(s, j+2) \end{array}$$

2. if s[j+3] = E,

$$\begin{array}{ll} & S_1(s,j) \\ = & \mathit{loop}(\text{TREE}, [\,3{,}2{,}0\,], s, j, [\,\emptyset, \text{R}, \emptyset, \emptyset\,], \mathsf{BMop}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,2{,}0\,], s, j, [\,\emptyset, \text{R}, \emptyset, \text{E}\,], \mathsf{BMop}) \\ = & S_{1,3}(s,j) \end{array}$$

3. if s[j+2] ≠ E,

$$\begin{array}{ll} & S_{1,3}(s,j) \\ = & \mathit{loop}(\text{TREE}, [\,2{,}0\,], s, j, [\,\emptyset, \text{R}, \emptyset, \text{E}\,], \mathsf{BMop}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,2{,}0{,}1{,}3\,], s, \mathit{j{+}1}, [\,\text{R}, \{\text{E}\}, \text{E}, \emptyset\,], \mathsf{BMop}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,0{,}1{,}3\,], s, \mathit{j{+}1}, [\,\text{R}, \{\text{E}\}, \text{E}, \emptyset\,], \mathsf{BMop}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,1{,}0{,}3{,}2\,], s, \mathit{j{+}2}, [\,\{\text{E}\}, \text{E}, \emptyset, \emptyset\,], \mathsf{BMop}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,0{,}3{,}2{,}1\,], s, \mathit{j{+}3}, [\,\text{E}, \emptyset, \emptyset, \emptyset\,], \mathsf{BMop}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,3{,}2{,}1{,}0\,], s, \mathit{j{+}4}, [\,\emptyset, \emptyset, \emptyset, \emptyset\,], \mathsf{BMop}) \\ = & S(s, j+4) \end{array}$$

We may be tempted to conclude that the BMop strategy gives rise to the BMop algorithm ...

7.5.4 A strategy BMst to obtain BMst

We now look for a search strategy BMst such that *subs2* when partially evaluated wrt. fixed p and BMst implements the BMst algorithm.

First some notation: we say that an information array is *well-formed* if it *either* does not contain any information at all (i.e. all elements are No-info) *or* the last element is *not* No-info.

Then we can define BMst as follows:

- if *inf* is well-formed, then BMst behaves as BMop:

$$\mathsf{BMst}(\mathit{inf}) = \mathsf{Pos}^R \mathbin{++} \mathsf{Neg} \mathbin{++} \mathsf{Nothing}^R$$

- if *inf* is not well-formed, then

$$\mathsf{BMst}(\mathit{inf}) = \mathsf{Pos}^R \mathbin{++} \mathsf{Neg} \mathbin{++} [\,\mathsf{FORGET}\,] \mathbin{++} \mathsf{All}^R$$

Explained in words, what happens is: when a mismatch has been found *inf* will cease to be well-formed, as a Shift operation is made. After checking that the information in Pos and Neg does not make it possible to detect new mismatches, everything is forgotten about this information so the pattern string has to matched in its entirety against the subject string.

In order to argue that BMst indeed implements the BMst algorithm, we (as usual) consider the pattern string TREE and will show that *subs2*, when PE'd wrt. TREE and BMst, behaves as BMst in the examples from section 7.3.2 and section 7.3.3. It is not hard to see that this will be the case provided the target program contains the following rules, with S being the main function:

$S(s,j) \rightarrow S_3(s,j)$, `if` $s[j+3] = \text{E}$

$S(s,j) \rightarrow S(s,j+2)$, `if` $s[j+3] = \text{R}$

$S_3(s,j) \rightarrow S(s,j+4)$, `if` $s[j+2] = \text{T}$

$S_3(s,j) \rightarrow S(s,j+1)$, `if` $s[j+2] = \text{R}$

In order to show this, first let S and S_3 be abbreviations as defined below:

$S(s,j) = loop(\text{TREE},[3,2,1,0],s,j,[\emptyset,\emptyset,\emptyset,\emptyset],\mathsf{BMst})$

$S_3(s,j) = loop(\text{TREE},[2,1,0],s,j,[\emptyset,\emptyset,\emptyset,\text{E}],\mathsf{BMst})$

Then the claim follows from the calculations below:

1. if s[j+3] = E,

$$\begin{aligned}
& S(s,j) \\
= \ & loop(\text{TREE},[3,2,1,0],s,j,[\emptyset,\emptyset,\emptyset,\emptyset],\mathsf{BMst}) \\
\rightarrow \ & loop(\text{TREE},[2,1,0],s,j,[\emptyset,\emptyset,\emptyset,\text{E}],\mathsf{BMst}) \\
= \ & S_3(s,j)
\end{aligned}$$

2. if s[j+3] = R,

$$\begin{aligned}
& S(s,j) \\
= \ & loop(\text{TREE},[3,2,1,0],s,j,[\emptyset,\emptyset,\emptyset,\emptyset],\mathsf{BMst}) \\
\rightarrow \ & loop(\text{TREE},[2,\mathsf{FORGET},3,2,1,0],s,j+1,[\emptyset,\emptyset,\{\text{E}\},\emptyset],\mathsf{BMst}) \\
\rightarrow \ & loop(\text{TREE},[1,\mathsf{FORGET},3,2,1,0],s,j+2,[\emptyset,\{\text{E}\},\emptyset,\emptyset],\mathsf{BMst}) \\
\rightarrow \ & loop(\text{TREE},[\mathsf{FORGET},3,2,1,0],s,j+2,[\emptyset,\text{R},\emptyset,\emptyset],\mathsf{BMst}) \\
\rightarrow \ & loop(\text{TREE},[3,2,1,0],s,j+2,[\emptyset,\emptyset,\emptyset,\emptyset],\mathsf{BMst}) \\
= \ & S(s,j+2)
\end{aligned}$$

3. if s[j+2] = T,

$$\begin{array}{ll} & S_3(s,j) \\ = & \mathit{loop}(\text{TREE}, [\,2,1,0\,], s, j, [\,\emptyset,\emptyset,\emptyset,\text{E}\,], \mathsf{BMst}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,2,1,\mathsf{FORGET},3,2,1,0\,], s, j+1, [\,\emptyset,\{\text{E}\},\text{E},\emptyset\,], \mathsf{BMst}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,1,\mathsf{FORGET},3,2,1,0\,], s, j+1, [\,\emptyset,\{\text{E}\},\text{E},\emptyset\,], \mathsf{BMst}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,1,0,\mathsf{FORGET},3,2,1,0\,], s, j+2, [\,\{\text{E},\text{R}\},\text{E},\emptyset,\emptyset\,], \mathsf{BMst}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,0,\mathsf{FORGET},3,2,1,0\,], s, j+3, [\,\text{E},\emptyset,\emptyset,\emptyset\,], \mathsf{BMst}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,3,2,1,0\,], s, j+4, [\,\emptyset,\emptyset,\emptyset,\emptyset\,], \mathsf{BMst}) \\ = & S(s, j+4) \end{array}$$

4. if s[j+2] = R,

$$\begin{array}{ll} & S_3(s,j) \\ = & \mathit{loop}(\text{TREE}, [\,2,1,0\,], s, j, [\,\emptyset,\emptyset,\emptyset,\text{E}\,], \mathsf{BMst}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,2,1,\mathsf{FORGET},3,2,1,0\,], s, j+1, [\,\emptyset,\{\text{E}\},\text{E},\emptyset\,], \mathsf{BMst}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,1,\mathsf{FORGET},3,2,1,0\,], s, j+1, [\,\emptyset,\{\text{E}\},\text{E},\emptyset\,], \mathsf{BMst}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,\mathsf{FORGET},3,2,1,0\,], s, j+1, [\,\emptyset,\text{R},\text{E},\emptyset\,], \mathsf{BMst}) \\ \rightarrow & \mathit{loop}(\text{TREE}, [\,3,2,1,0\,], s, j+1, [\,\emptyset,\emptyset,\emptyset,\emptyset\,], \mathsf{BMst}) \\ = & S(s, j+1) \end{array}$$

We may be tempted to conclude that the BMst strategy gives rise to the BMst algorithm . . .

7.5.5 Discussion

- The observant reader will notice that no search strategies corresponding to the BMor/BMna algorithms were given – as those algorithms exploit their information in a rather unsystematic way, it is hard to see how to obtain them by PE of *subs2*. An analogous observation is made in [PS88], where BMst is derived by formal methods from a naive substring matcher. The authors explain that the reason why they have arrived at (what we have called) BMst instead of BMor is that their derivation employs purely formal reasoning.
- The definition of the search strategy BMst may seem contrived, requiring a substantial amount of insight (or rather hindsight. . .).

On the other hand, the definition can be viewed in the light of the desire to prevent the number of specialized functions to explode: hence the information array has to be "cleaned" from time to time (motivating the FORGET element); and in order to avoid a lot of specialized versions of *loop* with identical values of *inf* but different values of *iseq* one has to start "from scratch" afterwards (motivating why the same element may occur more than once).

- A historical remark: by investigating the natural strategy BMop I discovered BMop *before* learning that Knuth had been 14 years ahead...

7.6 Related work

In [Dyb85] it is investigated how to specialize Earley's general context free parser wrt. a fixed grammar; the conclusion is that a lot of handwork is necessary.

In [CD89] (this paper being the starting point of the work reported in this chapter) the KMP algorithm is derived by PE, after a naive substring matcher has been transformed into one more suitable for PE purposes. Contrary to the claims of the paper, this transformation cannot (in my opinion!) be considered automatic; neither is it obvious that it preserves semantics.

In [Sun90] some very efficient variations of the Boyer-Moore idea are presented (but not within the context of program derivation). The approach is characterized by two features:

- After a mismatch has been found in e.g. the situation

  ```
  ..TEX...
     ^
  TREE
  ```

  it is the symbol in the subject string *immediate to the right* of the right end of the pattern string, i.e. the X, which (partly) determines how far to shift (while e.g. in BMor it is the T that would (partly) determine the shift). It should be clear that in this way shifts will be longer in average, and hence the matcher faster. It is not clear whether it will be possible to express this idea within the framework of partially evaluating a (one-stage) substring matcher.

- Most interestingly, the paper employs the idea of a parametrized search strategy. It is shown how other choices than the left-to-right and the right-to-left may be (statistically) better, the search sequence being dependent on the actual value of the pattern string: for instance, the pattern character occurring less frequently in the alphabet could be tested first.

The idea of maintaining a description of what is known about the input is employed in [Jør90], where an interpreter for a language with pattern matching is rewritten into one suitable for PE (thus a compiler can be generated).

The development in [QG91] is rather similar to ours: first a pattern language is defined, where patterns can be built up using constructors like `cons`, `quote`, `or`, `any` (with the obvious meanings) etc. – also recursive patterns are allowed. A pattern matcher is presented, which carries around a description of what is known about the input. This matcher can be partially evaluated wrt. known pattern, yielding the KMP algorithm – and if the arguments to `cons` are processed in reversed order, BMop is obtained.

Last, but certainly not least, one must acknowledge the work done by Olivier Danvy within the field – unfortunately mostly unpublished, but being a great inspiration and covering a wide range of topics, e.g. how to get the effect of Weiner trees (cf. [AHU74, section 9.5]).

Chapter 8

A model for a logic language

In this chapter we will formalize the idea of multilevel transition systems in a *logic programming setting.* The main purpose this time will be to give sufficient conditions for *total correctness* of unfold/fold transformations, such that (most of) the results from the literature emerge as special cases (after the frameworks in question have been encoded into our framework). Compared with chapter 4, the major differences (besides shifting from a functional view into a logic view) will be that

- we only consider 2-level systems;
- we model folding *explicitly* (instead of viewing foldings as abbreviations);
- we do not formulate speedup theorems.

Giving conditions for total correctness of unfold/fold transformations is a rather hot topic in the logic programming community, see e.g. [TS84], [KK90], [Sek91], [GS91], [PP91a], [BCE92]. This is unlike the situation in the functional community where Kott's work (cf. section 4.9) seems rather isolated. It is tempting to explain this difference by observing that

- operational semantics traditionally enjoy a more respectable status in the former community than in the latter[1];
- operational semantics better capture the essence of unfolding and folding: unfolding corresponds to a transition being made in the "right" direction, folding corresponds to a transition being made

[1]Recently, for instance the attempts to integrate functional programming and concurrency (see e.g. [LG88], [Nie89], [BMT92]) have renewed the interest in operational semantics for functional languages.

in the “wrong” direction. By using a denotational approach, this cannot be expressed directly (cf. the claims p. 12 and p. 123).

Not only total correctness, but also the existence of speedup bounds and the concept of ultimate sharing were crucial facets of our functional model, as developed in the previous chapters. Accordingly, let us now briefly (before giving an overview of this chapter) digress on how those two issues are carried over to the logic world:

8.0.1 Speedup bounds in the logic world

We certainly can expect something similar to theorem 4.6.3 and theorem 4.6.4 to hold – in fact, these two theorems were originally stated for a model of a logic language (without explicit folding) [Amt91]. Also we can, similar to what is done in section 4.8, factor out some features whose presence enables more than a constant speedup. Of special interest is the feature discussed in section 4.8.1 (i.e. the level 1 evaluation order being non-optimal), since

1. it is common in logic languages (as e.g. in PROLOG) to employ the strategy always to unfold the leftmost goal (this strategy will be denoted $\mathcal{LR}$);
2. the $\mathcal{LR}$ strategy is not optimal;
3. hence a large speedup may be possible if “some non-$\mathcal{LR}$ steps are made during transformation”.

The technique of letting the transformation “simulate” an (optimal) evaluation strategy is presented in [BDSK89] (and further elaborated in e.g. [DSMSB90]) where it is called *compiling control.* The scope of this technique is quite wide, an important special case being when the source program has been designed by means of the “generate and test” paradigm (first all candidate solutions are generated; then it is tested whether they really are solutions). As stated e.g. [SS86, p. 207] the key to improve efficiency in such cases is to “push” the tester inside the generator as “deep” as possible, making it possible to discard failed candidates before they are fully generated.

As a simple example of this, consider the task of producing a list of length N containing b's only. If the alphabet in question is $\{a, b\}$, a naive (and silly!) way to proceed is expressed by the program below, where N

is coded as a unary number (in the following we assume that *sol* is called with N instantiated and with L uninstantiated):

$sol(N,L) \leftarrow gen(N,L),\ test(L)$

$gen(0,[\,]) \leftarrow \Box;\ gen(\mathsf{s}(N),[\mathrm{a}|L]) \leftarrow gen(N,L);\ gen(\mathsf{s}(N),[\mathrm{b}|L]) \leftarrow gen(N,L)$

$test([\,]) \leftarrow \Box;\ test([\mathrm{b}|L]) \leftarrow test(L)$

This is clearly *exponential* in N when evaluated using the $\mathcal{LR}$ strategy. On the other hand, *linear* complexity is obtained if one uses a strategy where $test(L)$ is unfolded as soon as the first element of L becomes instantiated. It is possible to produce a target program which "simulates" this strategy; first consider the goal $sol(0,L)$. This is unfolded to $gen(0,L),test(L)$; then *gen* is unfolded binding L to [ ] giving $test([\,])$, and finally $test([\,])$ is unfolded – we thus have produced the clause

$sol(0,[\,]) \leftarrow \Box$

Next consider the goal $sol(\mathsf{s}(N),L)$, which we initially can unfold into $gen(\mathsf{s}(N),L),test(L)$. As two clauses for *gen* match, we split into two cases:

- by unfolding using the second clause for *gen*, L is bound to $[\mathrm{a}|L']$ giving the goal sequence $gen(N,L'),test([\mathrm{a}|L'])$. Now we unfold *test*; as no clauses match this branch fails.
- by unfolding using the third clause for *gen*, L is bound to $[\mathrm{b}|L']$ giving the goal sequence $gen(N,L'),test([\mathrm{b}|L'])$. Now unfold *test*; this gives the goal sequence $gen(N,L'),test(L')$ which can be folded back into $sol(N,L')$. We thus have produced the clause

  $sol(\mathsf{s}(N),[\mathrm{b}|L']) \leftarrow sol(N,L')$

For a more realistic example consider the following program, which sorts a list by generating all permutations and then all but the sorted one are discarded (we assume that *sort* is called with X instantiated and Y uninstantiated):

$sort(X,Y) \leftarrow perm(X,Y),ord(Y)$

$perm([\,],[\,]) \leftarrow \Box;\ perm([A|X],[B|Y]) \leftarrow del(B,[A|X],Z),\ perm(Z,Y)$

$del(A,[A|X],X) \leftarrow \Box;\ del(A,[B|X],[B|Y]) \leftarrow del(A,X,Y)$

$ord([\,]) \leftarrow \Box;\ ord([\,A\,]) \leftarrow \Box;\ ord([A,B|X]) \leftarrow A < B,\ ord([B|X])$

This program is clearly exponential in the length of X. Now, by performing some steps which are not $\mathcal{LR}$ steps we arrive at the following program (isomorphic to the one given [BDSK89, p. 140]) where *permord(A,X,Y)* is "an abbreviation" of *perm(X,Y),ord([A|Y])*:

sort([ ],[ ]) $\leftarrow\Box$

sort([A|X],[B|Y]) $\leftarrow$*del(B,[A|X],Z), permord(B,Z,Y)*

permord(A,[ ],[ ]) $\leftarrow\Box$

permord(A,[B|X],[C|Y]) $\leftarrow$*del(C,[B|X],Z),A* $<$ *C,permord(C,Z,Y)*

Even though efficiency has been improved, complexity is still exponential. In order to get a polynomial algorithm one has to do as in [TS84, p. 135]:

1. (re)define *perm* as follows:

   perm([ ],[ ]) $\leftarrow\Box$*; perm([A|X],Y)* $\leftarrow$*perm(X,Z), ins(A,Z,Y)*
   ins(A,X,[A|X]) $\leftarrow\Box$*; ins(A,[B|X],[B|Y])* $\leftarrow$*ins(A,X,Y)*

2. exploit the "law" (cf. section 4.8.3) that

   ins(A,Z,Y),ord(Y) is "equivalent" to ord(Z),ins(A,Z,Y),ord(Y).

   Then we can unfold *sort([A|X],Y)* (via *perm(X,Z),ins(A,Z,Y),ord(Y)*) into

   perm(X,Z),ord(Z),ins(A,Z,Y),ord(Y)

   hence we are justified in storing the clause

   sort([A|X],Y) $\leftarrow$*sort(X,Z),ins(A,Z,Y),ord(Y)*

   giving rise to a *cubic* target program.

Note that exploiting laws is *not* a subcase of compiling control, cf. the observation [BDSK89, p. 136]: "no computation rule provides lemma generation".

8.0.2 Ultimate sharing in the logic world

In the functional world it is always "safe" to look at a more general expression and unfold it "as far as possible" (theorem 5.1.7 provides the formal justification for this claim). Intuitively, this is due to a functional program returning *one* and *only one* answer. On the other hand, consider the logic program

p(b) $\leftarrow$*p(b)*

and the goal *p(a)* which of course should fail. However, if one considers the "more general" goal *p(X)* and unfolds it as far as possible, evaluation never terminates.

Of course one may exhibit some conditions under which it is safe to do "ultimate sharing" – we can expect that these conditions to a large degree amount to saying that the logic program in question is a "translation" of a functional program ...

8.0.3 A two-level transition system

After the digression above we return to the issue of setting up a two-level system for a logic language, with the purpose of reasoning about total correctness. As we want to model folding explicitly, it is helpful to impose some extra structure on the set of level 1 transitions (for formal definitions in the form of inference rules, see section 8.4):

- that t is a level 1 unfolding step from B to B' intuitively means that B' can be derived by unfolding one of the atoms in B, using a rule in $\mathcal{R}_0$;
- that t is a level 1 unfolding from B to B' means that B' can be derived from B by a sequence of level 1 unfoldings steps;
- that t is a level 1 folding step from B to B' intuitively means that B' can be derived from B by applying a rule in $\mathcal{R}_0$ "backwards";
- that t is a level 1 folding from B to B' means that B' can be derived from B by a sequence of level 1 foldings steps;
- that t is a level 1 transition from B to B' means that B' can be derived from B by a sequence of level 1 unfolding steps and level 1 folding steps.

The interpretation is that level 1 unfoldings model standard evaluation of the source program; whereas level 1 transitions model transformation of the source program. Accordingly, the level 1 rules (constituting the target program) are chosen among the set of level 1 transitions. Moreover, we have:

- that t is a level 2 unfolding step from B to B' intuitively means that B' can be derived from B by unfolding one of the atoms in B, using a rule in $\mathcal{R}_1$;
- that t is a level 2 unfolding from B to B' means that B' can be derived from B by a sequence of level 2 unfolding steps.

8.0.4 An overview of this chapter

The aim of section 8.1 will be to give the reader a flavor of the main features of our model. In particular,

- in section 8.1.1 we present the basic intuition of our approach and introduce the concept of *U-mirrors*, a representation of (the control part of) an unfold/fold transformation which facilitates reasoning about preservation of termination properties;
- in section 8.1.2 we discuss how to ensure total correctness, wrt. various evaluation strategies;
- in section 8.1.3 we focus upon the data aspect, which is usually modeled by means of substitutions – we will propose an alternative approach;
- in section 8.1.4 we discuss when it is permissible to fold against a given clause – not wrt. total correctness, but wrt. partial correctness;
- in section 8.1.5 we discuss how to extend the model such that it is able to represent the whole search tree and not only a single branch.

Section 8.1 will be rather informal, based on examples and intuition. All concepts introduced will be formally defined and all theorems will be proved in the subsequent sections. Section 8.2 compares with related work.

In section 8.3, the basic machinery is set up, e.g. concerning configurations, transitions and U-mirrors. In section 8.4, a two-level transition system is defined. In section 8.5, we state and prove various theorems concerning sufficient conditions for total correctness. In section 8.6 the whole story is repeated, transitions now representing search trees instead of single branches – here some proofs will appeal rather heavily to intuition, but of course these may be formalized at the expense of decreased clarity.

First, however, we give a "realistic" example of the unfold/fold technique:

Example 8.0.1 Consider the source program

f([ ],[ ]) $\leftarrow\Box$

*f([N|U],[*s*(N)|V])* $\leftarrow$*f(U,V)*

g(X,Z) $\leftarrow$*f(X,Y),f(Y,Z)*

Operationally, *f* adds one to each element in a list of unary numbers. Thus *g* will traverse its input list *X* twice. Our aim will be to make a target program where *g* only traverses its input list once: first consider the configuration *g([ ],Z)*. This can be unfolded into the configuration *f([ ],Y),f(Y,Z)*. By unfolding the first *f*, *Y* gets bound to [ ] and we arrive at the configuration *f([ ],Z)*. Now this *f* can be unfolded, binding *Z* to [ ]. We are thus able to let the target program contain the rule

$$g([\,],[\,]) \leftarrow\Box \tag{8.1}$$

Next consider the configuration *g([N|X],Z)*. This can be unfolded into *f([N|X],Y),f(Y,Z)*. By unfolding the first *f*, *Y* gets bound to *[*s*(N)|Y1]* and we get the configuration *f(X,Y1),f([*s*(N)|Y1],Z)*. By unfolding the second *f*, *Z* gets bound to *[*s*(*s*(N))|Z1]* and we arrive at the configuration *f(X,Y1),f(Y1,Z1)*. As *Y1* is *a new unbound variable*, this can now be folded back into the configuration *g(X,Z1)*. We are thus able to let the target program contain the rule

$$g([N|X],[\mathsf{s}(\mathsf{s}(N))|Z1]) \leftarrow g(X,Z1) \tag{8.2}$$

Now consider the "query" *g([0,0],Z)*. If the target program is used to "solve" this query, it is first rewritten into *g([0],Z1)* binding *Z* to *[*s*(*s*(0))|Z1]*; then rewritten into *g([ ],Z2)* binding *Z1* to *[*s*(*s*(0))|Z2]* and finally rewritten into the empty configuration, binding *Z2* to [ ]. Thus the query is solved using three inference steps, and Z has been bound to $[\,\mathsf{s}(\mathsf{s}(0)),\ \mathsf{s}(\mathsf{s}(0))\,]$.

It is easily seen that the same query, $g([\,0,0\,],Z)$, also can be solved with the same binding to Z by using the source program – but then seven inference steps are needed. □

8.1 An outline of the theory

8.1.1 Modeling control

It should be quite obvious that the transformation in example 8.0.1 preserves termination properties. One way of arguing for this is to observe that the first argument to g gets "smaller"[2] for each inference step (assuming that g is called with a first argument which is fully instantiated). However, we would rather like a way of reasoning which only depends on the syntactic structure of the transformation process. As promised p. 125, our approach will be to generalize Kott's insight: that the number of unfoldings should exceed the number of foldings. Below we give the basic intuition why this works, only focusing on the control part and abstracting away the data part.

Our aim is to show that if the target program loops, then the source program loops as well. So suppose the target program loops. This means that there is a level 2 unfolding sequence of form

$$g \rightarrow g \rightarrow g \rightarrow \dots \tag{8.3}$$

Since each level 1 rule of form $g \rightarrow g$ corresponds of a level 1 transition of form

$$g \rightarrow f\,f \rightarrow f\,f \leftarrow g$$

(that is one unfolding, two "parallel" unfoldings and one folding) the level 2 unfolding sequence in (8.3) corresponds to a level 1 transition sequence

$$g \rightarrow f\,f \rightarrow f\,f \leftarrow g \rightarrow f\,f \rightarrow f\,f \leftarrow g \rightarrow f\,f \rightarrow f\,f \dots \tag{8.4}$$

A key point now is that a folding is "canceled" by a subsequent unfolding, that is $f\,f \leftarrow g \rightarrow f\,f$ is "equivalent" to the empty sequence. By applying this canceling process to (8.4), we end up with a level 1 unfolding sequence

$$g \rightarrow f\,f \rightarrow f\,f \rightarrow f\,f \rightarrow f\,f \dots$$

[2]cf. the discussion p. 124.

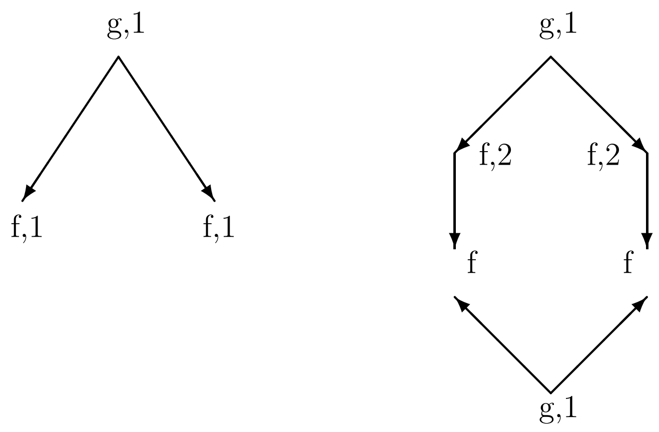


Figure 8.1: Two U-mirrors

which shows that the source program loops, as desired.

The notation employed above is rather ambiguous. For instance, we said that $f\,f \rightarrow f\,f$ denoted that a parallel unfolding of the two f's took place, but how to denote that only one of the f's is unfolded? To this end we introduce a device, to be called an *U-mirror*. An U-mirror is composed by two trees (or rather two collection of trees, such a collection to be termed an *U-forest*), the first representing the "unfolding part" of the transformation and the second representing the "folding part" of the transformation. Accordingly, the leaves of the first tree must equal the leaves of the second tree.

Let us use figure 8.1 as an illustrating example; here the U-mirrors corresponding to the level 1 rules (8.1) and (8.2) from example 8.0.1 are depicted. First consider the first U-mirror, which consists of the unfolding part only. The root of this tree is labeled "g,1" because initially g was unfolded using the *first* (and only) rule for g. Next the two occurrences of f were unfolded using the *first* rule for f – accordingly the two nodes labeled f are labeled with a 1 as well. It may appear as if the tree has leaves f and f, but since both of these are unfolded into the empty goal sequence the tree really has no leaves.

Next consider the second U-mirror, where the unfolding part as well as the folding part has two leaves: f and f. That two internal nodes are labeled "f,2" is because the two occurrences of f were unfolded (into f) when deriving the rule, in both cases using the *second* level 0 rule for f.

U-mirrors will be treated in depth in section 8.3.2. The use of U-mirrors (in my opinion!) facilitates reasoning about transitions, enabling e.g. the Church-Rosser completion to be expressed as a pushout in a suitable category (cf. lemma 8.4.3).

8.1.2 Conditions for total correctness

We have seen how to represent transformations by U-mirrors. Our goal will be to give conditions on these U-mirrors which guarantee total correctness.

Weights

Our starting point was the intuition: if there are "more unfoldings than foldings", transformation is safe. In order to get more widely applicable conditions this has to be generalized, taking into account that "some unfoldings are more important than others". To see this, consider the program

E(a) $\leftarrow$*A*

E(b) $\leftarrow$*B*

E(X:Y) $\leftarrow$*E(X),E(Y)*

A $\leftarrow$*B,B*

...

Starting with *E(a)*, we can unfold this into *A* and further into *B,B*. This can be folded back into *B,E(b)* into *E(b),E(b)* and finally folded back into *E(b:b)*, yielding a target program clause

E(a) $\leftarrow$*E(b:b)*

As two unfolding steps and three folding steps have been made, the reasoning technique from section 8.1.1 cannot be used to show total correctness of the transformation. However, we can argue that the clause above represents some progress in the computation process, as *A* is unfolded into *B,B* but never folded back. This can be formalized by assigning *weights* (non-negative numbers) to the arcs in the U-mirrors representing a transition, such that the weight of an arc is a function of the predicate symbol being unfolded.[3] We can now define the weight of a path in a U-mirror as the sum of the weights encountered when walking along the

[3]Actually, the weight may also depend on which rule is used and which conjunct the arc represents.

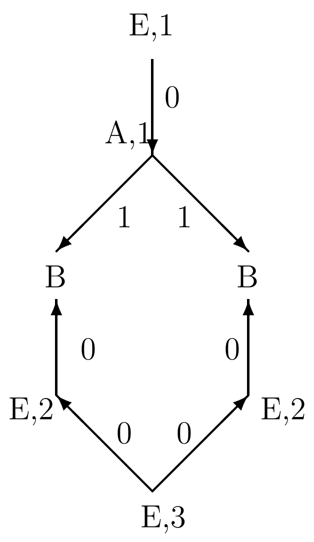


Figure 8.2: A U-mirror with weights

path, where the weights of arcs in the folding part are negated before contributing to the summation.

By assigning arcs from E weight 0 and arcs from A weight 1, the target program clause above is represented by the U-mirror depicted in figure 8.2. We see that all paths have weights 1 – but if we had assigned E weight 2 all paths would have weight -1.

Conditions wrt. various semantics

Whether a given transformation preserves termination properties depends on which semantics is chosen, that is which evaluation strategy is employed at level 1 (and level 2). We shall concentrate upon two kind of strategies:

- a *fair* strategy, which loosely speaking is one which sooner or later unfolds any goal;
- the $\mathcal{LR}$ strategy.

Concerning total correctness wrt. a fair strategy, we have the following

Condition 8.1.1 Suppose weights can be assigned in a way such that it for all level 1 rules holds that all the paths in the corresponding U-mirror have weight ≥ 1. Suppose B loops at level 2 by a fair strategy. Then B loops at level 1 by a fair strategy too. (This is theorem 8.5.2.) $\square$

Concerning total correctness wrt. the $\mathcal{LR}$ strategy, we have

Condition 8.1.2 Suppose weights can be assigned in a way such that it for all level 1 rules holds that the leftmost path in the corresponding U-mirror has weight ≥ 1. Suppose B loops at level 2 by the $\mathcal{LR}$ strategy. Then B loops at level 1 by the $\mathcal{LR}$ strategy too. (This is theorem 8.5.3.) $\square$

On the other hand, if the transformation does some non-$\mathcal{LR}$ steps it may happen that the domain of termination is *increased.* To see this, consider the source program

p(X) $\leftarrow$*q(X),r(X); q(a)* $\leftarrow$*q(a); r(b)* $\leftarrow\square$

Starting with the configuration *p(X)*, this can be unfolded into *q(X),r(X)* and then by a *non-*$\mathcal{LR}$ unfolding into *q(b)*, yielding the target program

p(b) $\leftarrow$*q(b)*

Now *p(X)* terminates (and fails) at level 2 by any strategy, while *p(X)* loops at level 1 by the $\mathcal{LR}$ strategy.

The same source program shows that it may happen that a transformation is total correct wrt. the $\mathcal{LR}$ strategy but not wrt. a fair strategy: again starting with the configuration *p(X)* we unfold this into *q(X),r(X)* and then we unfold the leftmost atom yielding *q(a),r(a).* This can be folded back into *p(a)*, yielding the target program

p(a) $\leftarrow$*p(a)*

It is easily seen that this transformation is total correct wrt. the $\mathcal{LR}$ strategy – *p(t)* loops at level 2 (by the $\mathcal{LR}$ strategy) iff t can be unified with *a* iff *p(t)* loops at level 1 by the $\mathcal{LR}$ strategy. This is as predicted by condition 8.1.2, since it is possible to assign weights in a way (e.g. 1 to q and 0 to p) such that the leftmost path of the U-mirror corresponding to this transformation has weight ≥ 1.

On the other hand, *p(X)* loops at level 2 (by any strategy) but terminates at level 1 by a fair strategy. Thus the transformation is not total correct wrt. a fair strategy.

Having defined the weight of a U-mirror as the sum of the weights occurring in it, the weights occurring in the folding part negated, we can formulate a – less useful – condition:

Condition 8.1.3 Suppose weights can be assigned in a way such that it for all level 1 rules holds that the corresponding U-mirror has weight

≥ 1. Suppose B loops at level 2 by some strategy. Then B also loops at level 1, by some strategy. (This is theorem 8.5.1.) $\Box$

This condition is not enough to guarantee total correctness (neither wrt. fair nor $\mathcal{LR}$ semantics): consider the source program *p(X)* $\leftarrow$*r(X),q(X)*; *q(a)* $\leftarrow$*q(a)*; *r(b)* $\leftarrow\Box$. By unfolding *p*; unfolding *q* and finally folding into *p* we get the level 1 rule *p(a)* $\leftarrow$*p(a)*. If *q* is assigned weight ≥ 1, the corresponding U-mirror will have weight ≥ 1. Now e.g. *p(a)* loops at level 2 by any strategy, but fails at level 1 by a fair strategy as well as by the $\mathcal{LR}$ strategy.

8.1.3 Modeling data

We have to settle on the form of the configurations. It seems clear that a configuration should be a sequence of predicate symbols, together with some information about which values the arguments to those predicates can assume. One usually represents this information as a substitution, as e.g. in [Llo84], but below we shall argue for choosing another representation.

A key point in the reasoning in section 8.1.1 is that a folding followed by an unfolding (of the same predicate symbol) should cancel each other, i.e. be equivalent to the identity. Let us see if this can be achieved by means of the substitution model. Consider a clause *p(X)* $\leftarrow$*q(X)* and let the initial configuration be (*q(X)*,ε) where ε is the identity substitution. We want to do a folding step; to this end we have to rename the clause yielding say *p(X1)* $\leftarrow$*q(X1)*. As a result of the folding we get the configuration (*p(X1)*,{*X1* $\rightarrow$ *X*}). We want to do an unfolding step; to this end we have to rename the clause yielding say *p(X2)* $\leftarrow$*q(X2)*. As a result of the unfolding we get the configuration (*q(X2)*,{*X1* $\rightarrow$ *X*, *X2* $\rightarrow$ *X*}). The substitution on the right hand side is "equivalent" to the identity substitution, in the sense that *X2* can assume any value, but some superfluous items have crept in. Of course it will be possible to repair on that, but this almost inevitably gets rather messy (as witnessed by various papers in the literature!), since substitutions are hard to reason about from an algebraic point of view (even though e.g. [Søn89] and [Pal89] show that certain sets of substitutions carry some structure), in particular one has to be very careful about renaming.

We therefore (as a first attempt!) shall prefer to represent data as *sets of ground values.* It is useful to suppose the existence of a universal data domain $\mathcal{D}$. We will impose no requirements on the structure of this set; in our examples, however, we shall assume the elements of $\mathcal{D}$ to be PROLOG ground terms, i.e. terms built inductively from some set of functors (constants just being zero-arity functors). For instance, the configuration (*p(X)*,ε) will be represented by the predicate symbol p together with the set of ground values $\{d|d \in D\}$ (that is, all values are allowed). The configuration (*p(X)*,$\{ X \rightarrow a \}$) will be represented by p together with the singleton set of ground values $\{a\}$. And the configuration (*p(X,Y)*, $\{ X \rightarrow Y \}$) is represented by p together with the set $\{(d,d)|d \in D\}$ (that is, only pairs where the components are equal are allowed). It turns out that this way of representation is not quite enough, but let us stay with it for a moment.

Next consider how to represent clauses. As a first example, we shall look at the clause

p(X,X) $\leftarrow$*q(X)*

The idea will be to represent this clause as a *mapping* ϕ from ground values into sets of ground values: as the call to p fails if the first argument does not equal the second argument, $\phi(d_1,d_2) = \emptyset$ if $d_1 \neq d_2$. Otherwise, we have $\phi(d,d) = \{d\}$.

From what is said it should be clear that we want to find a function ϕ^{-1} such that $\phi^{-1}\star\phi$ is the identity. And this is quite easy: choose $\phi^{-1}(d) = \{(d,d)\}$.

As a second example, consider a clause where a new variable occurs on the right hand side:

p(X) $\leftarrow$*q(X,Y)*

This clause is represented as a mapping ψ, where $\psi(d) = \{(d,d')|d' \in \mathcal{D}\}$. In this case, however, it is impossible to find a mapping ψ^{-1} such that $\psi^{-1}\star\psi$ yields the identity. On the other hand, one can define ψ^{-1} on certain subsets Z such that $\psi(\psi^{-1}(Z)) = Z$ – these subsets are those with the property that if they contain an element (d_1,d_2) then they contain any other element of form (d_1,d'_2). It is worth noticing that this phenomenon explains why one has to be careful when folding against a clause containing variables not occurring in the head; these variables must be "uninstantiated". This is a problem to which some incorrect solutions have been proposed in the literature (and yet proved correct!), for a survey see [GS91].

We want to ensure that all mappings to be defined by program clauses are reversible. Of course, there are many technically equivalent ways to proceed, but the one we shall adopt is to represent data as a family of sets of ground values, to be called an *information family.* As an example of this, consider again the rule *p(X)* $\leftarrow$*q(X,Y)*. Here the left hand side will be represented as a $\mathcal{D}$-indexed family Q where the d'th element is the singleton set $\{d\}$, whereas the right hand side will be represented as a $\mathcal{D}$-indexed family Q' where the d'th element is the set $\{(d,d')|d' \in \mathcal{D}\}$. The mapping ψ defined by the program clause now simply for each $d \in \mathcal{D}$ maps the d'th element in Q into the d'th element in Q', and is thus clearly reversible.

One will notice that the representation of a substitution is no longer unambiguous. For instance, *p(X)* equipped with the identity substitution may be represented by the $\mathcal{D}$-indexed family Q where for each $d \in \mathcal{D}$ it holds that $Q(d)$ is the singleton set $\{d\}$; but may also be represented by the singleton family Q' consisting of the set $\{d|d \in \mathcal{D}\}$. The difference is that the latter representation (as we saw above) is needed to cope with variables occurring on the right hand side only; but on the other hand cannot be “instantiated”.

This leads to our next point: when is the a configuration B an instance of another configuration B'? The “predicate part” of B and B' must be equal, so we only focus upon information families – assume B contains K-indexed information family Q and assume B' contains K'-indexed information family Q'. The answer turns out to be that there must exists a mapping s from K into $\mathcal{P}(K')$ such that for all $k \in K$, $Q(k) = \cup_{k' \in s(k)} Q'(k')$. As an example of this take $K = K' = \mathcal{D}$, $Q(a) = \{a\}$, $Q(d) = \emptyset$ for $d \neq a$, $Q'(d) = \{d\}$ for all $d \in \mathcal{D}$. As Q corresponds to the substitution $\{X \rightarrow a\}$ and Q' corresponds to the identity substitution, it seems clear that Q is to be considered an instance of Q'. And so it is according to our definition, as we can choose s as follows: $s(a) = \{a\}$, $s(d) = \emptyset$ otherwise. We say that $B = \mathcal{I}_s(B')$.

Configurations are not unfolded arbitrarily far, but only until either the sequence of predicate symbols is empty or the data part is *failure.* That a K-indexed information family Q is failure naturally means that $Q(k) = \emptyset$ for all $k \in K$.

We are now finished with our sketch of how configurations look. For a full account of configurations and operations on these, see section 8.3.1. Of course, one might ask about the precise relationship between our model and the “standard” model. Such an “equivalence result” would undoubtful be rather cumbersome to state and to prove, so we hope the reader is convinced that our model *is* a faithful representation of what is going on when a logic program is executed. Below we shall try to support this claim by working on a relatively (!) large example – this material might be skipped.

Our model in action

We shall use example 8.0.1 as our starting point. The level 0 rule *f([ ],[ ])* $\leftarrow\Box$ is represented as a transition from $B_1 = ([f\,], Q_1)$ to $B_1' = ([], Q_1')$, where Q_1 and Q_1' are $\mathcal{D}\times\mathcal{D}$-indexed families with $Q_1(d_1, d_2) = \{(d_1, d_2)\}$, $Q_1'([\,],[\,]) = \{()\}$, $Q_1'(d_1, d_2) = \emptyset$ if $d_1 \neq [\,]$ or $d_2 \neq [\,]$.

The level 0 rule *f([N|U],[*s*(N)|V])* $\leftarrow$*f(U,V)* is represented as a transition from $B_2 = ([f\,], Q_2)$ to $B_2' = ([f\,], Q_2')$ where the $\mathcal{D}\times\mathcal{D}$-indexed information families Q_2 and Q_2' are given by $Q_2(d_1, d_2) = \{(d_1, d_2)\}$, $Q_2'([d_n|d_u], [\mathsf{s}(d_n)|d_v]) = \{(d_u, d_v)\}$ and $Q_2'(d_1, d_2) = \emptyset$ if (d_1, d_2) is not of the form above.

In the standard framework, the query *f([0],Z)* is solved yielding an answer substitution where Z is bound to $[\,\mathsf{s}(0)\,]$. Now consider how this works in our framework. There the query *f([0],Z)* is represented as the configuration $B = ([f\,], Q)$ where the $\mathcal{D}$-indexed information family Q is given by $Q(d) = \{([\,0\,], d)\}$. Now consider the mapping s from $\mathcal{D}$ to $\mathcal{P}(\mathcal{D}\times\mathcal{D})$ given by $s(d) = Q(d)$. Then for all d it will trivially hold that

$$Q(d) = \bigcup_{(d_1,d_2)\in s(d)} Q_2(d_1, d_2)$$

The existence of this s shows that B is an instance of B_2, in our notation written $B = \mathcal{I}_s(B_2)$. Then there will be a level 1 unfolding step from B to $\mathcal{I}_s(B_2')$, to be denoted B'. $B' = ([f\,], Q')$ where Q' is a $\mathcal{D}$-indexed family given by

$$Q'(d) = \bigcup_{(d_1,d_2)\in s(d)} Q_2'(d_1, d_2)$$

That is, $Q'([\mathsf{s}(0)|d_v]) = \{([\,], d_v)\}$ and $Q'(d) = \emptyset$ otherwise.

Next consider the mapping s' from $\mathcal{D}$ to $\mathcal{P}(\mathcal{D}\times\mathcal{D})$ given by $s'(d) = Q'(d)$. Then for all d it will trivially hold that

$$Q'(d) = \bigcup_{(d_1,d_2)\in s'(d)} Q_1(d_1, d_2)$$

This means that $B' = \mathcal{I}_{s'}(B_1)$. Then there will be a level 1 unfolding step from B' to $\mathcal{I}_{s'}(B'_1)$, to be denoted B''. $B'' = ([], Q'')$ where Q'' is a $\mathcal{D}$-indexed family given by

$$Q''(d) = \bigcup_{(d_1,d_2)\in s'(d)} Q'_1(d_1, d_2)$$

That is, $Q''([\mathsf{s}(0)|d_v]) = Q'_1([\,], d_v)$ and $Q''(d) = \emptyset$ otherwise, i.e. $Q''([\,\mathsf{s}(0)\,]) = \{()\}$ and $Q''(d) = \emptyset$ otherwise. As $Q''(d) \neq \emptyset$ iff $d = [\,\mathsf{s}(0)\,]$, this corresponds to Z being bound to $[\,\mathsf{s}(0)\,]$ in the standard model.

Of course, also $B' = \mathcal{I}_{s'}(B_2)$. So there also is a level 1 unfolding from B' to $\mathcal{I}_{s'}(B'_2) = B'''$, where $B''' = ([f\,], Q''')$. However, it is easily seen that $Q'''(d) = \emptyset$ for all d – hence B''' is a failure configuration. Thus the transition from B' to B''' represents a failure branch.

In our examples we will, for ease of exposition, often switch back and forth between the standard model and our model when it is the control aspect which has our primary interest.

8.1.4 Modeling folding

Let some predicate symbol G be given, and let the level 0 rules for G be of form $\{t_i | i \in I\}$, each t_i going from B to B_i. Here B contains goal sequence $[G]$ and K-indexed information family Q, and each B_i contains K-indexed information family Q_i.

Given $i \in I$. Now (cf. the definition in section 8.4.4) it will be possible to make a level 1 folding step from $\mathcal{I}_s(B_i)$ to $\mathcal{I}_s(B)$ – where s is a mapping from K' to $\mathcal{P}(K)$, where $\mathcal{I}_s(B_i)$ contains K'-indexed information family Q'_i and where $\mathcal{I}_s(B)$ contains K'-indexed information family Q' – provided

1. for all $k' \in K'$, $Q'_i(k') = \emptyset$ iff $Q'(k') = \emptyset$;
2. $\mathcal{I}_s(B_{i'})$ is failure for $i' \neq i$;
3. B_i consists of a non-empty goal sequence.

The rationales for the above requirements are as follows:

1. It must not be possible to make a folding step from a failure configuration into a non-failure configuration. To see why, consider the two program clauses:

   p(a) ←q(a); q(X) ←q(X)

Starting with the configuration *p(X)*, one may consider unfolding it into *q(a)*, then unfold it once more into *q(a)*, and finally (erroneously!) fold back into *p(X)* – thus deriving the target program *p(X)* $\leftarrow$*p(X)*. As two unfoldings and only one folding is made, the reasoning in section 8.1.1 may tempt us to believe that this transformation preserves termination properties. However, e.g. the goal *p(b)* loops at level 2 while it fails at level 1. This is because the infinite sequence of level 2 unfolding steps *p(b)* $\Rightarrow$ *p(b)* $\Rightarrow \ldots$ corresponds to the sequence of level 1 unfold/fold steps where *p(b)* is unfolded into failure which then is unfolded into failure which then is folded back to *p(b)* etc.

Due to requirement 1, in our model it is not possible to make a folding step from a configuration whose information family contains *one* non-empty element only (as *q(a)*) into a configuration whose information family contains *many* non-empty elements (as *p(X)*).

2. This in the standard framework is modeled by the requirement that only one clause defining the predicate folded against should unify: if we have two program clauses

   p $\leftarrow$*q; p* $\leftarrow$*r*

   it must not be possible to fold *r* into *p* and then unfold into *q* – this would destroy semantics.

3. If there is a source program clause *p* $\leftarrow\Box$, it should not be possible to fold e.g. *q* into *q,p*. Such foldings never occur in practice, and it is convenient to exclude them: otherwise we above could derive the target program *p* $\leftarrow$*p*, and then *p* would loop at level 2 but as the corresponding level 1 transition unfolds *p* into *[]* which then is folded back into *p* etc, *p* does not loop at level 1.

8.1.5 Modeling the full search tree

So far a transition – for ease of exposition – only represents a single branch of the search tree, the transition system thus being non-confluent. In order to model the full search tree, configurations have to be *multisets* of "old" configurations (now to be called *basic configurations*). There are two reasons for working with multisets and not with sequences (i.e. not to order the branches), a pragmatic and a mathematical one:

- it is rather easy to implement or-parallelism [Gre87], as no communication has to occur between the branches. On the other hand, and-parallelism [Gre87] is much harder to implement due to the need for sharing of data, hence most implementations employ the $\mathcal{LR}$ strategy.

- If we use sequences, the Church-Rosser property will be lost. To see this, consider the program

  a ←b; a ←c; d ←e; d ←f

  Now consider the goal *[a,d]*. By first unfolding *a* and then unfolding *d* we first get *[b,d]; [c,d]* and then B_1 = *[b,e]; [b,f]; [c,e]; [c,f]*. By first unfolding *d* and then unfolding *a* we first get *[a,e]; [a,f]* and then B_2 = *[b,e]; [c,e]; [b,f]; [c,f]*. In [PP91a] one wants to distinguish between B_1 and B_2, and therefore unfolding of the *leftmost* atom only is allowed (unless extra conditions are satisfied.)

It is important to make the following observation: as configurations are multisets, backtracking is automatically accounted for in the model. On the other hand, when several (level 1) rules are applicable the choice between those is made *without* backtracking!

A configuration is said to be in normal form if all the basic configurations belonging to it are non-failure and with an empty goal sequence. Due to the Church-Rosser property, it then for a (basic) configuration B makes sense to define $[\![\mathrm{B}]\!]_1$ as follows:[4] if there exists a C in normal form and a level 1 unfolding from B to C, $[\![\mathrm{B}]\!]_1 = C$. Otherwise, $[\![\mathrm{B}]\!]_1 = \perp$. In a similar vein, one can define $[\![\mathrm{B}]\!]_2$. By restricting the level 1 (2) unfoldings in question to be $\mathcal{LR}$, one can define $[\![\mathrm{B}]\!]_1^L$ ($[\![\mathrm{B}]\!]_2^L$). Now condition 8.1.1 and 8.1.2 can be restated (a rule may now be represented by several U-mirrors):

Condition 8.1.4 Suppose that for all level 1 rules, represented by U-mirrors $m_1 \ldots m_k$, it holds for all m_i that all paths in m_i have weight ≥ 1. Then for all B, $[\![\mathrm{B}]\!]_2 = [\![\mathrm{B}]\!]_1$. (This is theorem 8.6.28.) □

Condition 8.1.5 Suppose that for all level 1 rules, represented by U-mirrors $m_1 \ldots m_k$, it holds for all m_i that the leftmost path in m_i has

[4]Notice that we identify a program which returns some answers and then loops with one which loops without producing any answers.

weight ≥ 1. Then for all B, $[\![\mathrm{B}]\!]_2^L \geq [\![\mathrm{B}]\!]_1^L$ – notice that the domain of termination may be increased, as shown in section 8.1.2. (This is theorem 8.6.29.) $\Box$

For a more detailed treatment and for proofs, see section 8.6.

In one way, the expressive power is enhanced by working with the full search tree: we can fold a configuration containing several basic configurations back into a single basic configuration – resembling the process of converting a NFA into a DFA. As an example of this, consider the program

ab([ ]) ←□; ab([a|X]) ←ab(X); ab([b|X]) ←ab(X)

bc([ ]) ←□; bc([b|X]) ←bc(X); bc([c|X]) ←bc(X)

abc(X) ←ab(X); abc(X) ←bc(X)

Now consider the configuration *abc([ ])*. This is unfolded into *ab([ ]);bc([ ])* which by two unfoldings yield □;□. The configuration *abc([a|X])* is unfolded into *ab([a|X]);bc([a|X])* which by two unfoldings yield *ab(X)* (as the second basic configuration is unfolded into failure). In a similar vein, the configuration *abc([c|X])* is unfolded into *bc(X)*.

The interesting case is where we start with the configuration *abc([b|X])*. Then we unfold into *ab([b|X])*; *bc([b|X])*, two more unfoldings yield *ab(X)*; *bc(X)* and now this can be *folded back* into *abc(X)*. We have thus derived five new rules for *abc*:

abc([ ]) ←□; abc([ ]) ←□

abc([a|X]) ←ab(X); abc([c|X]) ←bc(X)

abc([b|X]) ←abc(X)

To the latter rule correspond *two* U-mirrors, depicted in figure 8.3.

8.2 Related work

In the literature on unfold/fold transformations in logic languages transformation typically proceeds in a "step by step fashion"; after a goal in the body of a clause has been unfolded the clause is *deleted* from the program and *replaced* by the clause resulting from the unfolding – this is the approach taken in e.g. [GS91], [KK90], [PP91a], [Sek91], [TS84]. As

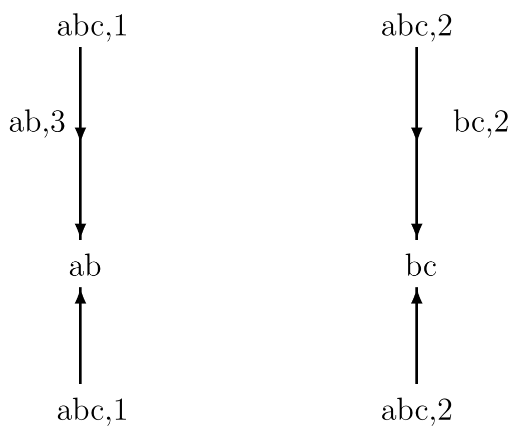


Figure 8.3: The two U-mirrors for *abc([b|X])* $\leftarrow$*abc(X)*

pointed out in [GS91], one by applying this method loses some power – to see this, consider the clause C = *p(f(X))* $\leftarrow$ *p(X)*. By our or similar techniques one is able to derive the clause C' : *p(f(f(f(X))))* $\leftarrow$ *p(X)* but this is impossible by the step-by-step method, since one – after having unfolded C against itself obtaining *p(f(f(X)))* $\leftarrow$ *p(X)* – has lost C. Aside from being less powerful, we also think that the step-by-step strategy conceptually is less clean than our approach – cf. the discussion p. 14.

In the literature, one is typically (contrary to our framework) not allowed to fold against a (direct or indirect) recursive predicate [KK90], [PP91a], [Sek91], [TS84]. This mirrors the view that folding corresponds to abbreviation, a view also held in [Amt91].

[TS84] and [KK90] divide the predicates into two classes: the *new* (corresponding to "eureka-definitions") and *old*, where folding is allowed against new predicates only. In the body of new predicates as well as in the body of old predicates, only old predicates can occur. Folding is valid in two cases:

- Starting with the definition of an old predicate, $O \leftarrow O_1 \ldots O_n$, one can do zero or more unfoldings of some of the O_i's and then fold some of these back into a new predicate.
- Starting with the definition of a new predicate, $N \leftarrow O_1 \ldots O_n$, one has to do *at least one* unfolding of some of the O_i's before folding back into a new predicate. [5]

[5]Actually, in [TS84] one is allowed to fold even if no unfolding of an O_i is made, provided not all the O_i's disappear by the folding. By assigning new predicates a weight equal the number of goals on the right hand side of their definition, and by assigning old predicates a "very large integer" as weight, this translates into our condition 8.1.3.

If new predicates are assigned weight 0 and old predicates are assigned weight 1, this translates into our condition 8.1.3. As we have seen in section 8.1.2 this condition is (too) weak, since failing branches may convert to loops.

[Sek91] improves on the above, essentially by coming up with condition 8.1.1 (still when new predicates have been assigned weight 0 and old predicates weight 1). As now not only the success set but also the failure set is preserved, negation can be handled as well.

[GS91] allows folding against *existing* clauses (recall clauses are deleted after having been unfolded) only (not allowing a clause to be folded against itself). This greatly limits the applications, since it seems impossible to arrive at recursive definitions of eureka-predicates. On the other hand, it becomes possible to give a relatively simple proof of termination preservation.

In contrast to the authors mentioned so far, [PP91a] impose an order on a sequence of goals, i.e. consider PROLOG's $\mathcal{LR}$ strategy. The crucial condition on folding is that the leftmost atom has been unfolded. Again by assigning the predicates folded against weight 0 and the others 1, the essence of this translates into our condition 8.1.2. A version of condition 8.1.2 is also stated in [Amt91].

[BCE92] gives sufficient conditions for *replacement* (folding being a special case) to be safe. The underlying intuition is perhaps best presented by means of their example 4: let the source program

c_1: $m(X) \leftarrow n(X)$

c_2: $n(0) \leftarrow \square$

c_3: $n(\mathsf{s}(X)) \leftarrow n(X)$

be given and consider the following two ways of transforming the clause c_3:

1. exploiting that we have the "equivalence"

$$m(X) \equiv n(X) \tag{8.5}$$

   we can replace $n(X)$ by $m(X)$, yielding the new clause

   c_3': $n(\mathsf{s}(X)) \leftarrow m(X)$

2. exploiting that we have the "equivalence"

$$m(\mathsf{s}(X)) \equiv n(X) \tag{8.6}$$

we can replace $n(X)$ by $m(\mathsf{s}(X))$, yielding the new clause

c_3'': $n(\mathsf{s}(X)) \leftarrow m(\mathsf{s}(X))$

It is easily seen that transformation 1 preserves termination properties, while transformation 2 introduces an infinite loop.

- In the framework of [BCE92] this behavior is explained as follows: the "dependency degree" of m wrt. c_3 is 1, as m in one step unfolds to something matching $n(\mathsf{s}(X))$. The equivalence (8.5) represents a "semantic delay" of 1, as $m(X)$ in one step unfolds to $n(X)$; whereas the equivalence (8.6) represents a semantic delay of 2, as it takes two steps for $m(\mathsf{s}(X))$ to unfold to $n(X)$. Now [BCE92, theorem 13] states that a sufficient condition for safeness is that the dependency degree is greater than or equal to the semantic delay.
- In our framework, the behavior is explained as follows: by e.g. assigning[6] m weight 1 and n weight 2 condition 8.1.1 tells us that it is safe[7] to fold c_3 against c_1, i.e. to do the replacement indicated by (8.5). On the other hand, the equivalence (8.6) represents *two* unfolding steps (first m is unfolded and then n), and now it is not possible to assign weights in a way such that condition 8.1.1 is satisfied.

Actually, example 5 in [BCE92] shows that often their condition amounts to the condition given in [Sek91] (and thus to our condition 8.1.1).

8.3 Fundamental concepts

We now embark on exhibiting the theory, an outline of which was presented in section 8.1. The material is highly technical, but contains no "deep" or surprising results. Some proofs (especially at the end of this chapter) are rather sketchy, if full proofs were to be given many more pages would be needed...

[6]It would not suffice to assign m weight 0 and n weight 1, as then the (unique path in the) U-mirror corresponding to the unchanged clause c_1 will have weight 0.

[7]Now the (unique paths in the) U-mirrors corresponding to c_1 and c_3' will have weight 1; as c_3 has weight 2 this reflects that evaluation has been "slowed down".

8.3.1 Basic configurations

The intuition behind the main definitions in this section was presented in section 8.1.3.

Assume a finite universe of predicate symbols U.

Definition 8.3.1 A *goal sequence* is a pair (J, H), where J is a totally ordered set and where H is a mapping from J into U. $\Box$

Often we drop J and just write H. $j < j'$ models that $H(j)$ is "to the left" of $H(j')$.

Definition 8.3.2 A *basic configuration* (over a set K) is a quadruple (J, H, K, Q) where (J, H) is a goal sequence and where Q is a mapping which to each $k \in K$ assigns a member of $\mathcal{P}(\prod_{j \in J} \mathcal{D})$ (for simplicity, we assume that all predicates have arity 1).

A basic configuration is *failure* if $Q(k) = \emptyset$ for all $k \in K$; and is *empty* if $J = \emptyset$. $\Box$

Definition 8.3.3 Given goal sequence (J, H), we define the *ca*nonical basic configuration over (J, H) as follows: $Ca_{(J,H)} = (J, H, \prod_{j \in J} \mathcal{D}, , Q)$ where $Q(\vec{d}) = \{\vec{d}\}$. $\Box$

Specializations

Definition 8.3.4 Given basic configurations B and B', with $B = (J, H, K, Q)$ and $B' = (J, H, K', Q')$. A *specialization* from B to B' [8] is a mapping s from K to $\mathcal{P}(K')$ such that for all $k \in K$

$$Q(k) = \bigcup_{k' \in s(k)} Q'(k')$$

We say that $B = \mathcal{I}_s(B')$. $\Box$

Fact 8.3.5 Given basic configuration $B = (J, H, K, Q)$. Now there exists one and only one specialization s from B to $Ca_{(J,H)}$. $\Box$

Proof: s will be a specialization iff

$$Q(k) = \bigcup_{\vec{d} \in s(k)} \{\vec{d}\} = s(k)$$

$\Box$

[8] In contrast to the functional model, this now is to be interpreted as saying that it is B' which is "more general" than B.

Operators on configurations and specializations

Definition 8.3.6 If J_1 and J_2 are two ordered sets (ordered by $<_1$ and $<_2$), we define $J = J_1 \& J_2$ (ordered by $<$) by letting J be the disjoint union of J_1 and J_2; by letting $\mathsf{in}_1(j) < \mathsf{in}_1(j')$ iff $j <_1 j'$ and $\mathsf{in}_2(j) < \mathsf{in}_2(j')$ iff $j <_2 j'$; and by letting $\mathsf{in}_1(j) < \mathsf{in}_2(j')$ for all $j \in J_1$, $j' \in J_2$. □

Definition 8.3.7 Let (J_1, H_1) and (J_2, H_2) be two goal sequences. We define $(J_1, H_1)\&(J_2, H_2)$ $(= (J, H))$ as follows: $J = J_1\&J_2$; $H(\mathsf{in}_1(j)) = H_1(j_1)$ and $H(\mathsf{in}_2(j)) = H_2(j)$. □

Definition 8.3.8 Let $B_1 = (J_1, H_1, K_1, Q_1)$ and $B_2 = (J_2, H_2, K_2, Q_2)$ be basic configurations. Then we define $B_1\&B_2 = (J, H, K, Q)$ as follows:

- $(J, H) = (J_1, H_1)\&(J_2, H_2)$.
- $K = K_1 \times K_2$
- $Q(k_1, k_2) = \{\vec{d}_1 \times \vec{d}_2 | \vec{d}_1 \in Q_1(k_1), \vec{d}_2 \in Q_2(k_2)\}$ where

$$(\vec{d}_1 \times \vec{d}_2)(\mathsf{in}_1(j)) = \vec{d}_1(j), (\vec{d}_1 \times \vec{d}_2)(\mathsf{in}_2(j)) = \vec{d}_2(j)$$

□

Fact 8.3.9 $B_1\&B_2$ is failure iff B_1 is failure or B_2 is failure. □

Definition 8.3.10 Given specializations s_1 from B_1 to B_1' and s_2 from B_2 to B_2'. Let $B_1 = (J_1, H_1, K_1, Q_1)$, $B_2 = (J_2, H_2, K_2, Q_2)$, $B_1' = (J_1, H_1, K_1', Q_1')$ and $B_2' = (J_2, H_2, K_2', Q_2')$. Then define $s = s_1\&s_2$, a specialization from $B_1\&B_2$ to $B_1'\&B_2'$, by

$$(s_1\&s_2)(k_1, k_2) = \{(k_1', k_2') | k_1' \in s_1(k_1), k_2' \in s_2(k_2)\}$$

□

We have to check that this actually *is* a specialization: but with $B_1\&B_2$ $= (J, H, K, Q)$ and $B_1'\&B_2' = (J, H, K', Q')$ we have

$$\begin{aligned}
Q(k_1, k_2) &= \{\vec{d}_1 \times \vec{d}_2 | \vec{d}_1 \in Q_1(k_1), \vec{d}_2 \in Q_2(k_2)\} \\
&= \{\vec{d}_1 \times \vec{d}_2 | \exists k_1' \in s_1(k_1), \exists k_2' \in s_2(k_2) : \vec{d}_1 \in Q_1'(k_1'), \vec{d}_2 \in Q_2'(k_2')\} \\
&= \{\vec{d} | \exists k_1' \in s_1(k_1), \exists k_2' \in s_2(k_2) : \vec{d} \in Q'(k_1', k_2')\} \\
&= \{\vec{d} | \exists k' \in s(k_1, k_2) : \vec{d} \in Q'(k')\} \\
&= \bigcup_{k' \in s(k_1, k_2)} Q'(k')
\end{aligned}$$

Definition 8.3.11 Given specialization s from B to B', and specialization s' from B' to B'', we define $s{\star}s'$, a specialization from B to B'', by (here $B = (J, H, K, Q)$, $B' = (J, H, K', Q')$, $B'' = (J, H, K'', Q'')$)

$$(s{\star}s')(k) = \bigcup_{k' \in s(k)} s'(k')$$

We have to check that this actually *is* a specialization:

$$Q(k) = \bigcup_{k' \in s(k)} Q'(k') = \bigcup_{k' \in s(k)} \bigcup_{k'' \in s'(k')} Q''(k'') = \bigcup_{k'' \in (s{\star}s')(k)} Q''(k'')$$

□

Definition 8.3.12 Given basic configuration $B = (J, H, K, Q)$ we define id_B, a specialization from B to B, by

$$\mathrm{id}_B(k) = \{k\}$$

□

Algebraic identities

When writing "=", we always mean "modulo isomorphism". It should be obvious what it means for two basic configurations to be isomorphic.

Fact 8.3.13 By letting the objects be basic configurations and by letting the morphisms be specializations, we obtain a category. That is, $\star$ is associative and id_B is a neutral element for all B.

Moreover, & is a functor in this category – i.e. $\mathrm{id}_{B_1} \& \mathrm{id}_{B_2} = \mathrm{id}_{B_1 \& B_2}$, and $(s_1{\star}s_1') \& (s_2{\star}s_2') = (s_1 \& s_2){\star}(s_1' \& s_2')$.

Finally, & is associative and $Ca_{H_1} \& Ca_{H_2} = Ca_{H_1 \& H_2}$. □

Proof: The only nontrivial part is the relation between & and $\star$:

$$\begin{aligned}
& (k_1'', k_2'') \in ((s_1{\star}s_1') \& (s_2{\star}s_2'))(k_1, k_2) \\
\Leftrightarrow\ & k_1'' \in (s_1{\star}s_1')(k_1), k_2'' \in (s_2{\star}s_2')(k_2) \\
\Leftrightarrow\ & \exists k_1', k_2' : k_1' \in s_1(k_1), k_1'' \in s_1'(k_1'), k_2' \in s_2(k_2), k_2'' \in s_2'(k_2') \\
\Leftrightarrow\ & \exists (k_1', k_2') : (k_1', k_2') \in (s_1 \& s_2)(k_1, k_2), (k_1'', k_2'') \in (s_1' \& s_2')(k_1', k_2') \\
\Leftrightarrow\ & (k_1'', k_2'') \in ((s_1 \& s_2){\star}(s_1' \& s_2'))(k_1, k_2)
\end{aligned}$$

□

8.3.2 U-mirrors

The intuition behind U-mirrors was presented in section 8.1.1. For a formal definition, we need some assumptions:

- let a function OI which for each $G \in U$ returns a non-empty and finite index set $\mathit{OI}(G)$ be given;
- let a function AI which for each $G \in U$ and each $i \in \mathit{OI}(G)$ returns a finite index set $\mathit{AI}(G)$, equipped with a total order $<$, be given;
- let a function P which for each $G \in U$, $i \in \mathit{OI}(G)$ and $j \in \mathit{AI}(G, i)$ returns $P(G, i, j) \in U$ be given;
- let a function W be given, which for each $G \in U$, $i \in \mathit{OI}(G)$ and $j \in \mathit{AI}(G, i)$ returns a non-negative integer $W(G, i, j)$, and for each $G \in U$, $i \in \mathit{OI}(G)$ with $\mathit{AI}(G, i) = \emptyset$ returns a non-negative integer $W(G, i)$.

A source program will in a natural way give rise to functions OI, AI and P – returning to example 8.0.1, there e.g. $\mathit{OI}(f) = \{1, 2\}$ (or any two-element set); $\mathit{OI}(g) = \{1\}$ (or any one-element set), $\mathit{AI}(f, 1) = \emptyset$, $\mathit{AI}(g) = \{1, 2\}$ with $1 < 2$, $P(g, 1, 1) = P(g, 1, 2) = f$. On the other hand, the weight function W (cf. section 8.1.2) can be chosen arbitrarily (but with some care, if one wants to prove total correctness of a given transformation – however, if $\mathit{AI}(G, i) = \emptyset$ the value of $W(G, i)$ can be chosen arbitrarily large without risk).

U-forests

Definition 8.3.14 A *U-forest* from goal sequence (J, H) to goal sequence (J', H') is a J-indexed family of trees where

1. Nodes are labeled by a *goal label* G, $G \in U$. Some nodes are also equipped with an *or-direction* label i with $i \in \mathit{OI}(G)$, meaning that they have been unfolded. Accordingly all nodes not being leaves, and possibly also some leaves, have an or-direction label. Leaves having an or-direction label (corresponding to nodes being unfolded into $\square$) also have a *weight label* w, with $w = W(G, i)$.
2. Arcs are labeled by an *and-direction* label j and a *weight label* w. Distinct arcs going from the same node are labeled by distinct and-direction labels.

3. For all $j \in J$, the root of the j'th tree has goal label $H(j)$.
4. Let N be a node which has an or-direction label i, and which has goal label G. Then j will be the and-direction label of an arc going from N iff $j \in AI(G, i)$. The arcs from N inherit the ordering of $AI(G, i)$.
5. Let a be an arc from a node N, with goal label G and or-direction label i, to N'. With j the and-direction label and w the weight label of a, the goal label of N' is $P(G, i, j)$ and $w = W(G, i, j)$.
6. There is a total ordering among the leaves – and thus also among the paths, where a path starts at a root and ends at a leaf – determined in the "natural way" by the ordering on J and the ordering on the arcs leaving each node.
7. The sequence of leaves *not having an or-direction label*, together with their goal labels, is isomorphic to (J', H').

- A path ending in a leaf not having an or-direction label is termed *working*.
- A U-forest is *working* iff all paths are working.
- The weight of a path p, $W(p)$, is the sum of the weight labels encountered when walking along p.
- The weight of a U-forest f, $W(f)$, is the sum of the weight labels in f.

$\square$

A path being working just means that it does not represent an unfolding into $\square$.

Written more formally, a working path p in a U-forest from (J, H) to (J', H') is a sequence of the form

$$jG_0(i_1, j_1, w_1)G_1 \ldots (i_n, j_n, w_n)j'G_n(n \geq 0)$$

where $j \in J$, $G_0 = H(j)$, $j' \in J'$, $G_n = H'(j')$, $G_k = P(G_{k-1}, i_k, j_k)$ and $w_k = W(G_{k-1}, i_k, j_k)$ for $k = 1 \ldots n$.

A non-working path in a U-forest from (J, H) to (J', H') is a sequence of the form

$$jG_0(i_1, j_1, w_1)G_1 \ldots (i_n, j_n, w_n)G_n iw(n \geq 0)$$

where $j \in J$, $G_0 = H(j)$, $AI(G_n, i) = \emptyset$, $w = W(G_n, i)$, $G_k = P(G_{k-1}, i_k, j_k)$ and $w_k = W(G_{k-1}, i_k, j_k)$ for $k = 1 \ldots n$.

In the first case $W(p) = \sum_{k=1}^{n} w_k$; in the second case $W(p) = \sum_{k=1}^{n} w_k + w$.

Definition 8.3.15 If p is a working path in a U-forest from (J, H) to (J', H') of form $jGqj'G'$, and p' is a path in a U-forest from (J', H') to (J'', H'') of form $j'G'q'$, then we define $p \star p' = jGqG'q'$. □

Definition 8.3.16 Given U-forest f from (J, H) to (J', H') and U-forest f' from (J', H') to (J'', H''). We can now define $f \star f'$, a U-forest from (J, H) to (J'', H''), by "gluing" the two forests together in the obvious way. □

Observation 8.3.17 Given a path p'' in $f \star f'$. Two possibilities:

- p'' is a non-working path in f. Then p'' will be non-working in $f \star f'$ as well.
- There exists working path p in f and path p' in f' such that $p'' = p \star p'$. p'' will be working iff p' is. These p and p' are unique.

Conversely, if p' is a path in f' there exists exactly one (working) path p in f such that $p \star p'$ forms a path in $f \star f'$. If p is a working path in f, there exists at least one path p' in f' such that $p \star p'$ forms a path in $f \star f'$. □

Definition 8.3.18 Given goal sequence (J, H). $\mathrm{id}_{(J,H)}$ is now defined as the U-forest from (J, H) to (J, H), where all paths are of the form jG. □

Definition 8.3.19 Given U-forest f_1 from H_1 to H_1' and U-forest f_2 from H_2 to H_2'. Now define $f_1 \& f_2$, a U-forest from $H_1 \& H_2$ to $H_1' \& H_2'$, in the obvious way – i.e. the paths in $f_1 \& f_2$ will be the "disjoint union" of the paths in f_1 and the paths in f_2. □

Fact 8.3.20 By letting the objects be goal sequences and the morphisms be U-forests, one gets a category (with $\star$ as composition and $\mathrm{id}\_$ as identities). & is a functor in this category, and & is associative. □

Definition 8.3.21 Given a U-forest f_1 from H to H_1, and a U-forest f_2 from H to H_2. We say that (f_1', f_2', H') is a *completion* of (f_1, f_2) if f_1' is a U-forest from H_1 to H', f_2' is a U-forest from H_2 to H', and $f_1 \star f_1' = f_2 \star f_2'$. □

Pushouts

Observation 8.3.22 Given a U-forest f_1 from H to H_1, and a U-forest f_2 from H to H_2. Suppose (f_1, f_2) has a completion. Then there exists a completion (f_1', f_2', H') such that for all completions (f_1'', f_2'', H'') there exists a U-forest f from H' to H'' with $f_1' \star f = f_1''$, $f_2' \star f = f_2''$. Consequently, this completion is unique – we term (f_1', f_2', H') the *pushout* of (f_1, f_2).

Moreover, if f_1 is working then f_2' will be working. □

Fact 8.3.23 Taking pushouts is commutative in the following sense: given (f_1, f_1') and (f_2, f_2'), such that f_1' and f_2 are U-forests to the same goal sequence. Suppose (f_1, f_1') has pushout (f_3, f_3'), and suppose $(f_2 \star f_3', f_2')$ has pushout (f_4, f_4'). Now the situation is as in the left part of figure 8.4.

Then (f_2, f_2') will have a pushout, to be written (f_5, f_5'); and also $(f_1, f_1' \star f_5)$ will have a pushout, to be written (f_6, f_6') – as depicted in the right part of figure 8.4. Moreover,

$$f_3 \star f_4 = f_6, f_5' \star f_6' = f_4'$$

□

Proof: Standard categorical reasoning is applied: As $f_2 \star f_3' \star f_4 = f_2' \star f_4'$ we see that (f_2, f_2') in fact has a pushout (f_5, f_5'), and there exists unique f such that $f_5 \star f = f_3' \star f_4$, $f_5' \star f = f_4'$. Now

$$f_1 \star f_3 \star f_4 = f_1' \star f_3' \star f_4 = f_1' \star f_5 \star f$$

which shows that $(f_1, f_1' \star f_5)$ has a pushout (f_6, f_6'), and there exists f' such that $f_6 \star f' = f_3 \star f_4$, $f_6' \star f' = f$.

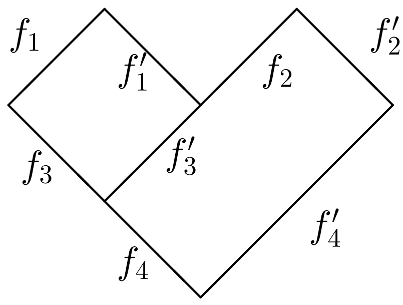
Also we have $f_5' \star f_6' \star f' = f_5' \star f = f_4'$. For reasons of symmetry (f_6 plays the same role as f_4', f_5' plays the same role as f_3 and f_6' plays the same role as f_4) we can find f'' such that $f_4' \star f'' = f_5' \star f_6'$, $f_3 \star f_4 \star f'' = f_6$.

From $f_6 = f_3 \star f_4 \star f'' = f_6 \star f' \star f''$ we conclude $f' = \mathrm{id}\_$ – hence the claim. □

U-mirrors

Definition 8.3.24 A *U-mirror* m from goal sequence H to goal sequence H' is a triple (f, f', H'')[9] where f is a U-forest from H to H''; and where

[9]We often just write (f, f') and omit H''.



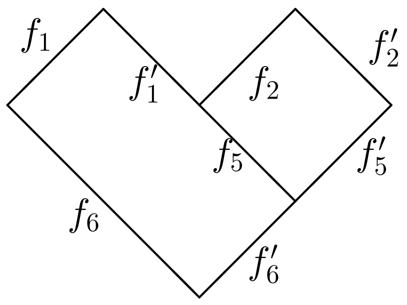


Figure 8.4: Pushout commutes

f' is a U-forest from H' to H''. We require f' to be working (as we do not allow folding using a rule of form $g \leftarrow \Box$).

- One can in the natural way define the paths of a U-mirror. A path p'' is *either* a non-working path in f *or* of the form (p, p') where p is a working path in f of form $qj''G''$ and p' a (working) path in f' of form $q'j''G''$ – we say p and p' are *connected*. Paths of the former kind are termed non-working; paths of the latter kind are termed working.
- We say that m is working iff all paths are working – i.e. iff f is working.
- The weight of a non-working path p, $W(p)$, is the weight of p in f; the weight of a working path (p, p'), $W(p, p')$, is the difference between the weight of p in f and the weight of p' in f'.
- The weight of the U-mirror, $W(m)$, is $W(f) - W(f')$.

$\Box$

Observe that given working path p in f there exists exactly one path p' in f' connected to p; and given (working) path p' in f' there exists exactly one path p in f connected to p'. Also, there will exist a working path in m iff H' is not empty.

We will not distinguish between a U-forest f from H to H' and the U-mirror $(f, \mathrm{id}_{H'}, H')$.

Definition 8.3.25 Let $m = (f, f', H'')$ be a U-mirror from H to H', and suppose m is working (i.e. f is working). Then we can define $\mathcal{R}(m)$, a (working) U-mirror from H' to H, by $\mathcal{R}(m) = (f', f, H'')$. $\Box$

Definition 8.3.26 Let $m_1 = (f_1, f_1', H_1'')$ be a U-mirror from H_1 to H_1', and let $m_2 = (f_2, f_2', H_2'')$ be a U-mirror from H_2 to H_2'. Now we can define $m_1 \& m_2$, a U-mirror from $H_1 \& H_2$ to $H_1' \& H_2'$, by letting

$$m_1 \& m_2 = (f_1 \& f_2, f_1' \& f_2', H_1'' \& H_2'')$$

□

Definition 8.3.27 Given goal sequence H. Then we can define id_H as $(\mathrm{id}_H, \mathrm{id}_H, H)$. □

Definition 8.3.28 Given a U-mirror $m_1 = (f_{11}, f_{21}, H_1')$ from H_1 to H_2, and given a U-mirror $m_2 = (f_{22}, f_{32}, H_2')$ from H_2 to H_3.

First suppose the pushout of (f_{21}, f_{22}) exists. Let it be (f_1', f_2', H''). Then

$$m_1 \star m_2 = (f_{11} \star f_1', f_{32} \star f_2', H'')$$

(This is a well-defined U-mirror: m_1 is a U-mirror, so f_{21} is working. By observation 8.3.22, f_2' is working. As m_2 is a U-mirror, f_{32} is working. Hence, $f_{32} \star f_2'$ is working.)

If the pushout does not exist, $m_1 \star m_2 = \perp$ (i.e. undefined). □

Lemma 8.3.29 Suppose $m_1 \star m_2 \neq \perp$. Suppose (p, p') is a path in $m_1 \star m_2$. Then there exists paths (p_1, p_1') in m_1, (p_2, p_2') in m_2, and paths p'', p''' such that $p = p_1 \star p''$, $p' = p_2' \star p'''$, and $p_1' \star p'' = p_2 \star p'''$. □

Proof: Let $m_1 = (f_1, f_1', (J_1', H_1'))$ be a U-mirror from (J_1, H_1) to (J_2, H_2), and let $m_2 = (f_2, f_2', (J_2', H_2'))$ be a U-mirror from (J_2, H_2) to (J_3, H_3). Let $(f'', f''', (J'', H''))$ be the pushout of (f_1', f_2). Then

$$m_1 \star m_2 = (f_1 \star f'', f_2' \star f''')$$

We have that p is a working path in $f_1 \star f''$, and p' is a (working) path in $f_2' \star f'''$. There thus exist working paths p_1 in f_1, p'' in f'', p_2' in f_2' and p''' in f''' such that $p = p_1 \star p''$, $p' = p_2' \star p'''$. Now let p_2 in f_2 and p_1' in f_1' be the unique paths connected to p_2' and p_1 respectively. As there exists $j_1' \in J_1'$ such that p_1 and p_1' both end with $j_1' H_1'(j_1')$, $p_1' \star p''$ will be a path in $f_1' \star f''$. Similarly, $p_2 \star p'''$ will be a path in $f_2 \star f'''$. Since $f_1' \star f'' = f_2 \star f'''$, and since $p_2 \star p'''$ ends with the same element in J'' as p''' does as p' does as p does as p'' does as $p_1' \star p''$ does, we conclude that actually $p_1' \star p'' = p_2 \star p'''$. □

Fact 8.3.30 1. Operating on U-mirrors, $\star$ is associative [10] with id_ as neutral element.

2. & is a functor; i.e. $(m_1 \star m_1') \& (m_2 \star m_2') = (m_1 \& m_2) \star (m_1' \& m_2')$; and $\mathrm{id}_H \& \mathrm{id}_{H'} = \mathrm{id}_{H \& H'}$. Also, & is associative.

3. The property of being working is closed under all operations in question.

4. Given working f from H to H'. Considered as a U-mirror, $\mathcal{R}(f) \star f = \mathrm{id}_{H'}$.

□

PROOF: First for the last claim: we must show that $(\mathrm{id}_{H'}, f) \star (f, \mathrm{id}_{H'}) = \mathrm{id}_{H'}$. Now the pushout of (f, f) is $(\mathrm{id}_{H'}, \mathrm{id}_{H'})$, hence the claim follows.

That id_ is neutral element can be seen as follows: Let $m = (f, f', H'')$ be a U-mirror from H to H'. The pushout of $(f', \mathrm{id}_{H'})$ clearly is $(\mathrm{id}_{H''}, f', H'')$. So

$$m \star \mathrm{id}_{H'} = (f \star \mathrm{id}_{H''}, \mathrm{id}_{H'} \star f', H'') = (f, f', H'') = m$$

That id_ is left neutral element is seen similarly.

Next for the associativity: let $m_1 = (f_1, f_1')$, $m_2 = (f_2, f_2')$, $m_3 = (f_3, f_3')$. Consider figure 8.5, where the relevant pushouts have been drawn (assuming they exist). Then

$$\begin{aligned} (m_1 \star m_2) \star m_3 &= (f_1 \star f_4 \star f_5, f_3' \star f_6') \\ m_1 \star (m_2 \star m_3) &= (f_1 \star f_6, f_3' \star f_4' \star f_5') \end{aligned}$$

Now we can apply fact 8.3.23 to see that $(m_1 \star m_2) \star m_3$ will be defined iff $m_1 \star (m_2 \star m_3)$ is, and then they will be equal.

The remaining claims are trivial. □

8.3.3 Properties of U-mirrors

Definition 8.3.31 Given n, we say that a U-mirror m satisfies $\mathcal{F}(n)$ iff for all working paths p we have $W(p) \geq n$. □

Lemma 8.3.32 If m_1 satisfies $\mathcal{F}(n_1)$, and m_2 satisfies $\mathcal{F}(n_2)$, then $m_1 \star m_2$ – if defined – satisfies $\mathcal{F}(n_1 + n_2)$. □

[10]Equality, in the presence of $\perp$, means that either both sides are $\perp$ or both sides are $\neq \perp$ and equal.

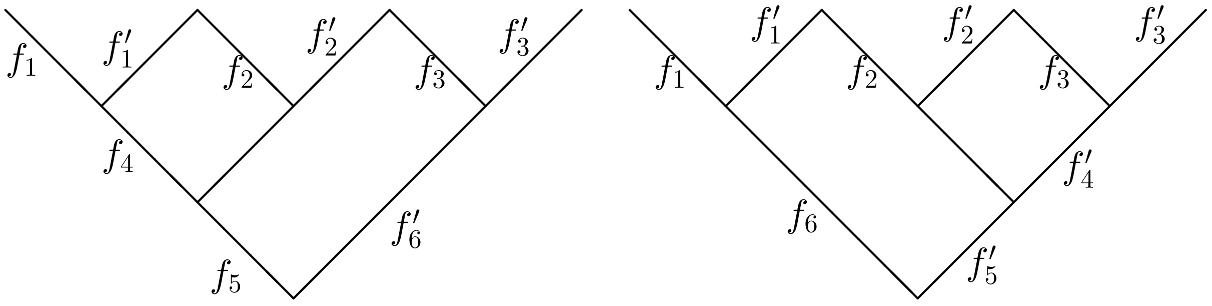


Figure 8.5: Associativity of $\star$ on U-mirrors

PROOF: Let $m_1 = (f_1, f'_1, H'_1)$ from H_1 to H_2; and let $m_2 = (f_2, f'_2, H'_2)$ from H_2 to H_3.

Let (p, p') be a working path in $m_1 \star m_2$. By lemma 8.3.29, there exists (p_1, p'_1) in m_1, (p_2, p'_2) in m_2 and paths p'', p''' such that $p = p_1 \star p''$, $p' = p'_2 \star p'''$, and $p'_1 \star p'' = p_2 \star p'''$. Now, by applying the assumption, we have

$$\begin{aligned} W(p,p') &= W(p) - W(p') = W(p_1) + W(p'') - W(p'_2) - W(p''') \\ &\geq W(p'_1) + n_1 + W(p'') - W(p_2) + n_2 - W(p''') \\ &= W(p'_1 \star p'') - W(p_2 \star p''') + n_1 + n_2 \\ &= n_1 + n_2 \end{aligned}$$

□

Definition 8.3.33 Given n, we say that a U-mirror m satisfies $\mathcal{A}(n)$ iff $W(m) \geq n$. □

Lemma 8.3.34 If m_1 satisfies $\mathcal{A}(n_1)$, and m_2 satisfies $\mathcal{A}(n_2)$, then $m_1 \star m_2$ – if defined – satisfies $\mathcal{A}(n_1 + n_2)$. □

PROOF: Let $m_1 = (f_1, f'_1, H'_1)$ from H_1 to H_2; and let $m_2 = (f_2, f'_2, H'_2)$ from H_2 to H_3. Let (f'', f''') be the pushout of (f'_1, f_2). Now

$$\begin{aligned} W(m_1 \star m_2) &= W(f_1 \star f'') - W(f'_2 \star f''') \\ &= W(f_1) + W(f'') - W(f'_2) - W(f''') \\ &\geq W(f'_1) + n_1 + W(f'') - W(f_2) + n_2 - W(f''') \\ &= W(f'_1 \star f'') - W(f_2 \star f''') + n_1 + n_2 \\ &= n_1 + n_2 \end{aligned}$$

□

Definition 8.3.35 Given n, we say that a U-mirror m satisfies $\mathcal{L}(n)$ iff there exists a path in m, and the leftmost path has weight $\geq n$. $\square$

Definition 8.3.36 Given a U-mirror m, we let

- $E(m)$ denote the number of non-working paths to the left, i.e. $E(m) \geq n$ iff the n leftmost paths in m are non-working.
- $L(m)$ denote the weight of the leftmost working path (if no such exists, 0).

Then we can define an ordering $\prec$ on the set of U-mirrors by letting $m_1 \prec m_2$ iff $E(m_1) < E(m_2)$ or $E(m_1) = E(m_2)$ and $L(m_1) < L(m_2)$. $\square$

Lemma 8.3.37 Suppose m_2 satisfies $\mathcal{L}(1)$, and $m = m_1 \star m_2$ is defined. Then $m_1 \prec m$. $\square$

Proof: First some notation: we say that a path is *left-directed* if the following holds for all its arcs: let the arc have and-direction label j and go from a node with goal label G and or-direction label i. Then j is the least element in $AI(G, i)$.

Let $m_1 = (f_1, f_1')$, $m_2 = (f_2, f_2')$. Let (f'', f''') be the pushout of (f_1', f_2). Now $m = (f_1 \star f'', f_2' \star f''')$. The $E(m_1)$ leftmost paths of m_1 will be non-working; so the $E(m_1)$ leftmost paths of f_1 will be non-working; so the $E(m_1)$ leftmost paths of $f_1 \star f''$ will be non-working; so the $E(m_1)$ leftmost paths of m will be non-working. This shows that $E(m_1) \leq E(m)$.

Now consider the leftmost path in m_2. Two possibilities:

- It is of form p_2 with p_2 non-working in f_2. As $f_2 \star f''' = f_1' \star f''$, there will exist p_1' in f_1' and left-directed p'' in f'' such that $p_2 = p_1' \star p''$, and such that p_1' is the leftmost path in f_1'. Let p_1' be connected to p_1; p_1 will be the leftmost working path in f_1. Now $p_1 \star p''$ will be non-working in $f_1 \star f''$ and hence also in m. This shows that $E(m_1) < E(m)$ and hence $m_1 \prec m$.
- It is of form (p_2, p_2') with p_2 working in f_2. As $f_2 \star f''' = f_1' \star f''$, there will exist p_1' in f_1', left-directed p'' in f'' and left-directed p''' in f''' such that $p_2 \star p''' = p_1' \star p''$, and such that p_1' is the leftmost path in f_1'. Let p_1' be connected to p_1; p_1 will be the leftmost working path

in f_1. Now $(p_1 \star p'', p'_2 \star p''')$ will belong to m, and be the leftmost working path in m. Moreover,

$$\begin{aligned}
L(m) &= W(p_1 \star p'') - W(p'_2 \star p''') \\
&= W(p_1) + W(p'') - W(p'_2) - W(p''') \\
&> W(p_1) + W(p'') - W(p_2) - W(p''') \\
&= W(p_1) + W(p'') - W(p_2 \star p''') \\
&= W(p_1) + W(p'') - W(p'_1 \star p'') \\
&= W(p_1) - W(p'_1) \\
&= L(m_1)
\end{aligned}$$

This shows $m_1 \prec m$.

□

8.3.4 Transitions

Definition 8.3.38 A *transition* t from basic configuration $B = (J, H, K, Q)$ to $B' = (J', H', K, Q')$ (notice the K-sets are identical) is a set of U-mirrors from (J, H) to (J', H') which is either empty or a singleton.[11] We will demand that (Q, Q') is a *non-increasing* pair, i.e. that for all $k \in K$, $Q(k) = \emptyset \Rightarrow Q'(k) = \emptyset$ (so it will not be possible to make a transition from a failure basic configuration into a non-failure, cf. the discussion in section 8.1.4, (1)).

We say that t is *stable* iff also (Q', Q) is a non-increasing pair, i.e. $Q(k) = \emptyset \Leftrightarrow Q'(k) = \emptyset$. □

Definition 8.3.39 Let t be a transition from B to B', and let t' be a transition from B' to B''. Now define $t \star t'$, a transition from B to B'', by

$$m'' \in t \star t' \Leftrightarrow \exists m \in t, m' \in t' : m'' = m \star m'$$

Clearly (B, B'') is a non-increasing pair, and $t \star t'$ will be stable if t and t' are stable. □

Definition 8.3.40 Let t_1 be a transition from B_1 to B'_1, and let t_2 be a transition from B_2 to B'_2. Now define $t_1 \& t_2$, a transition from $B_1 \& B_2$ to $B'_1 \& B'_2$, by

$$m \in t_1 \& t_2 \Leftrightarrow \exists m_1 \in t_1, m_2 \in t_2 : m = m_1 \& m_2$$

[11] We will often identify t with its element.

Clearly non-increasingness will be preserved, and $t_1 \& t_2$ will be stable if t_1 and t_2 are stable. $\square$

Definition 8.3.41 Given $B = (J, H, K, Q)$, define id_B, a transition from B to B, by letting id_B be the singleton set containing $\mathrm{id}_{(J,H)}$. This is clearly stable. $\square$

Definition 8.3.42 Given a transition t from B to B'. Let s be a specialization from $\mathcal{I}_s(B)$ to B. Now we define $\mathcal{I}_s(t)$, a transition from $\mathcal{I}_s(B)$ to $\mathcal{I}_s(B')$, by letting m be in $\mathcal{I}_s(t)$ iff m is in t. Clearly $(\mathcal{I}_s(B), \mathcal{I}_s(B'))$ is a non-increasing pair, and $\mathcal{I}_s(t)$ will be stable if t is. $\square$

Definition 8.3.43 Given a transition t from B to B'. Suppose t is stable, and suppose it for all $m \in t$ holds that m is working. Then we say that t is *reversible* and we can define $\mathcal{R}(t)$, a transition from B' to B, by stipulating that m is in $\mathcal{R}(t)$ iff $\mathcal{R}(m)$ is in t. Clearly $\mathcal{R}(t)$ is reversible. $\square$

Fact 8.3.44 By letting the objects be basic configurations and by letting the morphisms be transitions, we obtain a category. & is a functor in this category. $\square$

8.4 Two level transition system

This section formalizes the concepts from section 8.0.3. Further, we prove

- that unfolding at level 1 satisfies a *restricted form* of the Church-Rosser property (lemma 8.4.3); restricted since it (due to only one branch of the search tree being present) may happen that two unfoldings choose different branches, in which case we cannot hope for confluence –
- that an arbitrary sequence of unfolding/foldings may be replaced by a sequence of unfoldings followed by a sequence of foldings (lemma 8.4.15).

8.4.1 The level 0 rules

We now indicate how a source program gives rise to a set of level 0 rules: assume that for all $G \in U$ and $i \in OI(G)$ there exists a transition $t(G,i)$ from Ca_G to $c(G,i)$, where $c(G,i)$ takes the form

$$(AI(G,i), \lambda j.P(G,i,j), \mathcal{D}, Q)$$

and where $t(G,i)$ contains the U-mirror (U-forest) determined by the paths

$$\{G(i,j,\_)G' | j \in AI(G,i)\}$$

where $G' = P(G,i,j)$. However, if $AI(G,i) = \emptyset$ the U-mirror will be determined by the singleton path Gi. Clearly, this in non-increasing. Now

$$\mathcal{R}_0 = \{t(G,i) | G \in U, i \in OI(G)\} \tag{8.7}$$

In example 8.0.1, e.g. $c(g,1) = ([f,f], \mathcal{D} \times \mathcal{D}, Q)$ with

$$Q(d_1,d_2) = \{\{((d_1,d),(d,d_2))\} | d \in \mathcal{D}\}$$

8.4.2 Unfolding at level 1

We now define what it means for a transition t from B to B' to be a level 1 unfolding step, to be written $1 \vdash_u t\colon B \Rightarrow B'$[12] (here H_1 and H_2 are goal sequences, and s is a specialization):

$$\frac{t(G,i) \in \mathcal{R}_0}{1 \vdash_u \mathcal{I}_s(\mathrm{id}_{Ca_{H_1}} \& t(G,i) \& \mathrm{id}_{Ca_{H_2}})} \tag{8.8}$$

Next we define what it means for a transition t from B to B' to be a level 1 unfolding, to be written $1 \vdash_u^* t\colon B \Rightarrow B'$ [13]:

$$\frac{1 \vdash_u t\colon B \Rightarrow B'}{1 \vdash_u^* t\colon B \Rightarrow B'} \tag{8.9}$$

$$\frac{}{1 \vdash_u^* \mathrm{id}_B\colon B \Rightarrow B} \tag{8.10}$$

$$\frac{1 \vdash_u^* t\colon B \Rightarrow B', 1 \vdash_u^* t'\colon B' \Rightarrow B''}{1 \vdash_u^* t \star t'\colon B \Rightarrow B''} \tag{8.11}$$

If $1 \vdash_u^* t$, we can write $t = t_1 \star \ldots \star t_n$ ($n \geq 0$) with $1 \vdash_u t_i$ for $i \in \{1 \ldots n\}$. The least n which can be used is called the *length* of t.

[12] When we are not interested in the configurations, we may simply write $1 \vdash_u t$.

[13] When we are not interested in the configurations, we may simply write $1 \vdash_u^* t$.

Fact 8.4.1

1. If $1 \vdash_u t$, then $1 \vdash_u t\&\mathrm{id}_B$ and $1 \vdash_u \mathrm{id}_B\&t$ for basic configuration B.
2. If $1 \vdash_u t$, then $1 \vdash_u \mathcal{I}_s(t)$ for specialization s.
3. If $1 \vdash_u^* t$, then $1 \vdash_u^* t\&\mathrm{id}_B$ and $1 \vdash_u^* \mathrm{id}_B\&t$ for basic configuration B.
4. If $1 \vdash_u^* t_1$ and $1 \vdash_u^* t_2$, then $1 \vdash_u^* t_1\&t_2$.
5. If $1 \vdash_u^* t$, then $1 \vdash_u^* \mathcal{I}_s(t)$ for specialization s.
6. If $1 \vdash_u^* t$, then t is a singleton – i.e. of form $\{m\}$.

□

PROOF: For (1), notice that fact 8.3.5 tells us that we can write $B = \mathcal{I}_{s'}(Ca_H)$. Then

$$\begin{aligned}
& \mathcal{I}_s(\mathrm{id}_{Ca_{H_1}}\&t(G,i)\&\mathrm{id}_{Ca_{H_2}})\&\mathrm{id}_B \\
= {} & \mathcal{I}_s(\mathrm{id}_{Ca_{H_1}}\&t(G,i)\&\mathrm{id}_{Ca_{H_2}})\&\mathcal{I}_{s'}(\mathrm{id}_{Ca_H}) \\
= {} & \mathcal{I}_{s\&s'}(\mathrm{id}_{Ca_{H_1}}\&t(G,i)\&\mathrm{id}_{Ca_{H_2}}\&\mathrm{id}_{Ca_H}) \\
= {} & \mathcal{I}_{s\&s'}(\mathrm{id}_{Ca_{H_1}}\&t(G,i)\&\mathrm{id}_{Ca_{H_2\&H}})
\end{aligned}$$

(2) is trivial. (3) follows by induction in the derivation tree for $1 \vdash_u^* t$: the case where rule (8.9) has been applied follows from what has just been shown; the case where rule (8.10) has been applied is trivial; and the case where rule (8.11) has been applied follows from the calculation (cf. fact 8.3.44)

$$(t\&\mathrm{id}_B)\star(t'\&\mathrm{id}_B) = (t\star t')\&(\mathrm{id}_B\star\mathrm{id}_B) = (t\star t')\&\mathrm{id}_B$$

(4) follows from what has been just shown and fact 8.3.44:

$$t_1\&t_2 = (t_1\star\mathrm{id}\_)\&(\mathrm{id}\_\star t_2) = (t_1\&\mathrm{id}\_)\star(\mathrm{id}\_\&t_2)$$

(5) is a trivial induction in the derivation tree. (6) follows from the fact that all U-mirrors involved in fact are U-forests, thus $\star$ can never be undefined. □

The diamond lemma

Lemma 8.4.2 Suppose $1 \vdash_u t_1 \colon B \Rightarrow B_1$ and $1 \vdash_u t_2 \colon B \Rightarrow B_2$. Suppose t_1 and t_2 have a completion (viewed as U-forests). Then there exists B', transition t_1' from B_1 to B_1' and transition t_2' from B_2 to B_2' such that (t_1', t_2') (viewed as a U-forest) is the pushout of (t_1, t_2). Moreover, one of two holds:

- $B_1 = B_2 = B'$, $t_1 = t_2$, $t_1' = t_2' = \mathrm{id}_{B'}$.
- $1 \vdash_u t_1' \colon B_1 \Rightarrow B'$ and $1 \vdash_u t_2' \colon B_2 \Rightarrow B'$.

□

PROOF: We can assume that the transitions involved are of the following form:

$$\begin{aligned} t_1 &= \mathcal{I}_{s_1}(\mathrm{id}_{Ca_{H_{11}}} \& t(G_1, i_1) \& \mathrm{id}_{Ca_{H_{12}}}) \\ t_2 &= \mathcal{I}_{s_2}(\mathrm{id}_{Ca_{H_{21}}} \& t(G_2, i_2) \& \mathrm{id}_{Ca_{H_{22}}}) \end{aligned}$$

As t_1 and t_2 both are transitions from B, we get

$$\mathcal{I}_{s_1}(Ca_{H_{11} \& G_1 \& H_{12}}) = B = \mathcal{I}_{s_2}(Ca_{H_{21} \& G_2 \& H_{22}})$$

From this we infer $H_{11} \& G_1 \& H_{12} = H_{21} \& G_2 \& H_{22}$ and (by fact 8.3.5) that $s_1 = s_2$. Now two possibilities:

- $H_{11} = H_{21}$. Then $G_1 = G_2$, and $H_{12} = H_{22}$. As (t_1, t_2) has a completion, $i_1 = i_2$. This shows $t_1 = t_2$ and $B_1 = B_2$. Thus we can choose $t_1' = t_2' = \mathrm{id}_{B_1}$, and clearly (t_1', t_2') is the pushout.
- $H_{11} \neq H_{21}$. We can wlog. assume that H_{11} is shorter than H_{21}. Then there exists H such that $H_{12} = H \& G_2 \& H_{22}$, $H_{21} = H_{11} \& G_1 \& H$. Now define

  $$\begin{aligned} t_1' &= \mathcal{I}_{s_1}(\mathrm{id}_{Ca_{H_{11}}} \& \mathrm{id}_{c(G_1, i_1)} \& \mathrm{id}_{Ca_H} \& t(G_2, i_2) \& \mathrm{id}_{Ca_{H_{22}}}) \\ t_2' &= \mathcal{I}_{s_1}(\mathrm{id}_{Ca_{H_{11}}} \& t(G_1, i_1) \& \mathrm{id}_{Ca_H} \& \mathrm{id}_{c(G_2, i_2)} \& \mathrm{id}_{Ca_{H_{22}}}) \end{aligned}$$

  We can easily calculate

  $$\begin{aligned} t_1 \star t_1' &= \mathcal{I}_{s_1}(\mathrm{id}_{Ca_{H_{11}}} \& t(G_1, i_1) \& \mathrm{id}_{Ca_H} \& t(G_2, i_2) \& \mathrm{id}_{Ca_{H_{22}}}) \\ &= t_2 \star t_2' \end{aligned}$$

  To show that $1 \vdash_u t_1'$, notice that we can write $c(G_1, i_1)$ on the form $\mathcal{I}_{s'}(Ca_{H'})$ for some s',H' – similarly we have $1 \vdash_u t_2'$.

  Again, viewed as U-forests (t_1', t_2') is clearly the pushout.

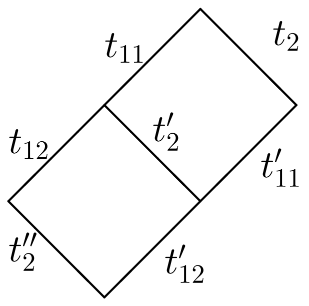


Figure 8.6: The Church-Rosser property

□

The Church-Rosser lemma

Lemma 8.4.3 Suppose $1 \vdash_u^* t_1\colon B \Rightarrow B_1$ and $1 \vdash_u^* t_2\colon B \Rightarrow B_2$. Suppose t_1 and t_2 have a completion, viewed as U-forests. Then there exists B' and transitions t'_1, t'_2 such that $1 \vdash_u^* t'_1\colon B_1 \Rightarrow B'$, $1 \vdash_u^* t'_2\colon B_2 \Rightarrow B'$, and such that (t'_1, t'_2) – viewed as U-forests – is the pushout of (t_1, t_2).

Moreover, the length of t'_1 is less than or equal the length of t_2, and the length of t'_2 is less than or equal the length of t_1. □

Proof: Induction in the length of t_1 plus the length of t_2. If t_1 or t_2 is the identity the claim is clear. If $1 \vdash_u t_1$ and $1 \vdash_u t_2$, the claim follows from lemma 8.4.2.

Otherwise, we can wlog. assume that $t_1 = t_{11} \star t_{12}$ with $1 \vdash_u^* t_{11}$, $1 \vdash_u^* t_{12}$. The situation is as depicted in figure 8.6. By induction, there exists t'_2, t'_{11} such that $1 \vdash_u^* t'_{11}$, $1 \vdash_u^* t'_2$ and such that (t'_2, t'_{11}) is the pushout of (t_{11}, t_2). Moreover, the length of t'_2 is less than or equal the length of t_2 – this enables us to use the induction hypothesis once more and arrive at t''_2, t'_{12} such that $1 \vdash_u^* t'_{12}$, $1 \vdash_u^* t''_2$.

Now we can choose $(t''_2, t'_{11} \star t'_{12})$ as the desired transition pair. That it actually is the pushout of (t_1, t_2) follows from fact 8.3.23. □

Lemma 8.4.3 shows that given a basic configuration $B = (J, H, K, Q)$ and an U-forest f from (J, H), it makes sense to define $\mathcal{U}_f(B)$ as the basic configuration B' such that $1 \vdash_u^* f\colon B \Rightarrow B'$.

8.4.3 Evaluation strategies and Looping at level 1

Definition 8.4.4 We say that B loops at level 1 *by some strategy* if for all $n \geq 1$ there exists a t_n and a B_n not failure such that $1 \vdash_u t_n\colon\ B_{n-1} \Rightarrow B_n$ (here $B_0 = B$). $\square$

Lemma 8.4.5 The following are sufficient conditions for B to loop at level 1 by some strategy:

1. For all $N \geq 0$, there for all n with $1 \leq n \leq N$ exists t_n and B_n not failure such that $1 \vdash_u t_n\colon\ B_{n-1} \Rightarrow B_n$ (again, $B_0 = B$).
2. For all $n \geq 0$, there exists B_n not failure and t_n s.t. $1 \vdash_u^* t_n\colon\ B \Rightarrow B_n$, where t_n viewed as a U-forest has n nodes with an or-direction label.
3. For all $n \geq 0$, there exists B_n not failure and t_n s.t. $1 \vdash_u^* t_n\colon\ B \Rightarrow B_n$, where t_n viewed as a U-forest has weight $\geq n$ – i.e. satisfies $\mathcal{A}(n)$.

$\square$

PROOF: That (1) implies that B loops at level 1 by some strategy is a consequence of Königs lemma (as there from a given B is a finite number of level 1 unfolding steps, since $OI(G)$ is finite). That (2) implies (1) is obvious. That (3) implies (2) is a consequence of the weights being upper bounded (as U, $OI(G)$ and $AI(G, i)$ are finite sets). $\square$

Fair strategy

Definition 8.4.6 We say that t is a *fair level 1 unfolding step* if t is of form $\mathcal{I}_s(r_1 \& \ldots \& r_k)$, $k \geq 1$, where $r_1 \ldots r_k$ are level 0 rules.

For a fair level 1 unfolding step t, we clearly have $1 \vdash_u^* t$. $\square$

Definition 8.4.7 We say that B loops at level 1 *by a fair strategy* if for all $n \geq 1$ there exists a fair level 1 unfolding step t_n and a B_n not failure such that $1 \vdash_u^* t_n\colon\ B_{n-1} \Rightarrow B_n$ (here $B_0 = B$). $\square$

Similar to lemma 8.4.5, we may prove

Lemma 8.4.8 The following are sufficient conditions for B to loop at level 1 by a fair strategy:

1. For all $N \geq 0$, there for all n with $1 \leq n \leq N$ exists a fair level 1 unfolding step t_n and B_n not failure nor empty such that $1 \vdash_u^* t_n\colon B_{n-1} \Rightarrow B_n$ (again, $B_0 = B$).

2. For all $n \geq 0$, there exists B_n not failure nor empty and t_n such that $1 \vdash_u^* t_n\colon B \Rightarrow B_n$, where each working path in t_n (viewed as a U-forest) has length $\geq n$.

3. For all $n \geq 0$, there exists B_n not failure nor empty and t_n such that $1 \vdash_u^* t_n\colon B \Rightarrow B_n$, where t_n satisfies $\mathcal{F}(n)$.

□

$\mathcal{LR}$ strategy

Definition 8.4.9 We say that t is a $\mathcal{LR}$ *level 1 unfolding step* if t is of form $\mathcal{I}_s(r\&\text{id\_})$ with r a level 0 rule.

We say that t is a $\mathcal{LR}$ *level 1 unfolding* if t is of form $t_1\star\ldots\star t_n$ with each t_i being a $\mathcal{LR}$ level 1 unfolding step.

The U-forest corresponding to a $\mathcal{LR}$ level 1 unfolding is termed a $\mathcal{LR}$ *U-forest.* □

Definition 8.4.10 We say that B loops at level 1 *by a $\mathcal{LR}$ strategy* if for all $n \geq 1$ there exists a $\mathcal{LR}$ level 1 unfolding step t_n and a B_n not failure such that $1 \vdash_u^* t_n\colon B_{n-1} \Rightarrow B_n$ (here $B_0 = B$). □

Recall the functions $E(m)$ and $L(m)$ introduced in section 8.3.3.

Lemma 8.4.11 The following are sufficient conditions for B to loop at level 1 by a $\mathcal{LR}$ strategy:

1. For all $N \geq 0$, there for all n with $1 \leq n \leq N$ exists a $\mathcal{LR}$ level 1 unfolding step t_n and B_n not failure such that $1 \vdash_u t_n\colon B_{n-1} \Rightarrow B_n$ (again, $B_0 = B$).

2. For all $n \geq 0$, there exists a transition t_n and B_n not failure, with $1 \vdash_u^* t_n\colon B \Rightarrow B_n$, such that $E(t_n) \geq n$.

3. For all $n \geq 0$, there exists a transition t_n and B_n not failure, with $1 \vdash_u^* t_n\colon B \Rightarrow B_n$, such that the length of the leftmost working path $\geq n$.

4. For all $n \geq 0$, there exists a transition t_n and B_n not failure, with $1 \vdash_u^* t_n\colon B \Rightarrow B_n$, such that $L(t_n) \geq n$.

□

Proof: First consider (1); then (2) ⇒ (1); then (3) ⇒ (1) and finally (4) ⇒ (3). □

8.4.4 Folding at level 1

We now define what it means for a transition t from B to B' to be a level 1 folding step, to be written $1 \vdash_f t\colon B \Rightarrow B'$.

We will assume the existence of a partial function $s(G,i)$ such that $\mathcal{I}_{s(G,i)}(t(G,i))$ is reversible, and such that $\mathcal{I}_{s(G,i)}(c(G,i'))$ is failure for all $i' \neq i$.

$$\frac{t(G,i) \in \mathcal{R}_0, s(G,i) \text{ defined}}{1 \vdash_f \mathcal{I}_s(\mathrm{id}_{Ca_{H_1}} \& \mathcal{R}(\mathcal{I}_{s(G,i)}(t(G,i))) \& \mathrm{id}_{Ca_{H_2}})} \tag{8.12}$$

Next we define what it means for a transition t from B to B' to be a level 1 folding, to be written $1 \vdash_f^* t\colon B \Rightarrow B'$:

$$\frac{1 \vdash_f t\colon B \Rightarrow B'}{1 \vdash_f^* t\colon B \Rightarrow B'} \tag{8.13}$$

$$\frac{}{1 \vdash_f^* \mathrm{id}_B\colon B \Rightarrow B} \tag{8.14}$$

$$\frac{1 \vdash_f^* t\colon B \Rightarrow B', 1 \vdash_f^* t'\colon B' \Rightarrow B''}{1 \vdash_f^* t{\star}t'\colon B \Rightarrow B''} \tag{8.15}$$

Fact 8.4.12 1. If $1 \vdash_f t$, then $1 \vdash_f t \& \mathrm{id}_B$ and $1 \vdash_f \mathrm{id}_B \& t$ for basic configuration B.

2. If $1 \vdash_f^* t_1$ and $1 \vdash_f^* t_2$, then $1 \vdash_f^* t_1 \& t_2$.

3. If $1 \vdash_f t$ ($1 \vdash_f^* t$), then $1 \vdash_f \mathcal{I}_s(t)$ ($1 \vdash_f^* \mathcal{I}_s(t)$) for specialization s.

4. If $1 \vdash_f t\colon B \Rightarrow B'$ (or $1 \vdash_f^* t\colon B \Rightarrow B'$), then t is reversible.

5. If $1 \vdash_f t\colon B \Rightarrow B'$ then $1 \vdash_u \mathcal{R}(t)\colon B' \Rightarrow B$. If $1 \vdash_f^* t\colon B \Rightarrow B')$ then $1 \vdash_u^* \mathcal{R}(t)\colon B' \Rightarrow B$.

□

PROOF: Mostly as in the proof of fact 8.4.1. The last point follows from

$$\begin{aligned}
& \mathcal{R}(\mathcal{I}_s(\mathrm{id}_{Ca_{H_1}} \& \mathcal{R}(\mathcal{I}_{s(G,i)}(t(G,i))) \& \mathrm{id}_{Ca_{H_2}})) \\
= \ & \mathcal{I}_s(\mathrm{id}_{Ca_{H_1}} \& \mathcal{I}_{s(G,i)}(t(G,i)) \& \mathrm{id}_{Ca_{H_2}}) \\
= \ & \mathcal{I}_s(\mathcal{I}_{\mathrm{id\_}}(\mathrm{id}_{Ca_{H_1}}) \& \mathcal{I}_{s(G,i)}(t(G,i)) \& \mathcal{I}_{\mathrm{id\_}}(\mathrm{id}_{Ca_{H_2}})) \\
= \ & \mathcal{I}_s(\mathcal{I}_{\mathrm{id\_} \& s(G,i) \& \mathrm{id\_}}(\mathrm{id}_{Ca_{H_1}} \& t(G,i) \& \mathrm{id}_{Ca_{H_2}})) \\
= \ & \mathcal{I}_{s \star (\mathrm{id\_} \& s(G,i) \& \mathrm{id\_})}(\mathrm{id}_{Ca_{H_1}} \& t(G,i) \& \mathrm{id}_{Ca_{H_2}})
\end{aligned}$$

□

8.4.5 Unfold/fold at level 1

We now define what it means for a transition t from B to B' to be a level 1 transition, to be written $1 \vdash^* t\colon B \Rightarrow B'$.

$$\frac{1 \vdash_u t\colon B \Rightarrow B'}{1 \vdash^* t\colon B \Rightarrow B'} \qquad (8.16)$$

$$\frac{1 \vdash_f t\colon B \Rightarrow B'}{1 \vdash^* t\colon B \Rightarrow B'} \qquad (8.17)$$

$$\frac{}{1 \vdash^* \mathrm{id}_B\colon B \Rightarrow B} \qquad (8.18)$$

$$\frac{1 \vdash^* t\colon B \Rightarrow B', 1 \vdash^* t'\colon B' \Rightarrow B''}{1 \vdash^* t \star t'\colon B \Rightarrow B''} \qquad (8.19)$$

Fact 8.4.13 1. If $1 \vdash^* t_1$ and $1 \vdash^* t_2$, then $1 \vdash^* t_1 \& t_2$.

2. If $1 \vdash^* t$, then $1 \vdash^* \mathcal{I}_s(t)$ for specialization s.

□

8.4.6 Fundamental properties of level 1 transitions

The switching lemma

Lemma 8.4.14 Suppose $1 \vdash_f t_1\colon B_1 \Rightarrow B$, $1 \vdash_u t_2\colon B \Rightarrow B_2$, with B_2 not failure. Then one of two holds:

- $B_1 = B_2$, $t_1 \star t_2 = \mathrm{id}_{B_1}$.

- There exists B', t'_1, t'_2 with $1 \vdash_u t'_1\colon\ B_1 \Rightarrow B'$, $1 \vdash_f t'_2\colon\ B' \Rightarrow B_2$ such that

  $$t_1 \star t_2 = t'_1 \star t'_2$$

$\square$

PROOF: This lemma, as well as its proof, is very similar to lemma 8.4.2.

We can assume that the transitions involved are of the following form:

$$\begin{aligned} t_1 &= \mathcal{I}_{s_1}(\mathrm{id}_{Ca_{H_{11}}} \& \mathcal{R}(\mathcal{I}_{s(G_1,i_1)}(t(G_1,i_1))) \& \mathrm{id}_{Ca_{H_{12}}}) \\ t_2 &= \mathcal{I}_{s_2}(\mathrm{id}_{Ca_{H_{21}}} \& t(G_2,i_2) \& \mathrm{id}_{Ca_{H_{22}}}) \end{aligned}$$

With $s = s_1 \star (\mathrm{id}\_ \& s(G_1,i_1) \& \mathrm{id}\_)$, we from the fact that t_1 is a transition to B and t_2 is a transition from B get

$$\mathcal{I}_s(Ca_{H_{11} \& G_1 \& H_{12}}) = B = \mathcal{I}_{s_2}(Ca_{H_{21} \& G_2 \& H_{22}})$$

From this we infer $H_{11} \& G_1 \& H_{12} = H_{21} \& G_2 \& H_{22}$ and that $s = s_2$. Now two possibilities:

- $H_{11} = H_{21}$. Then $G_1 = G_2$, and $H_{12} = H_{22}$. Again two possibilities:
  - $i_1 \neq i_2$. Then we have

    $$B_2 = \mathcal{I}_{s_1}(Ca_{H_{21}} \& \mathcal{I}_{s(G_1,i_1)}(c(G_1,i_2)) \& Ca_{H_{22}})$$

    but $\mathcal{I}_{s(G_1,i_1)}(c(G_1,i_2))$ is failure by the definition of $s(G,i)$, hence B_2 is failure contradicting our assumption.
  - $i_1 = i_2$. Then it is obvious that $B_1 = B_2$. And

    $$\begin{aligned} t_1 \star t_2 &= \mathcal{I}_s(\mathrm{id}_{Ca_{H_{11}}} \& (\mathcal{R}(t(G_1,i_1)) \star t(G_1,i_1)) \& \mathrm{id}_{Ca_{H_{12}}}) \\ &= \mathcal{I}_s(\mathrm{id}_{Ca_{H_{11}}} \& \mathrm{id}_{c(G_1,i_1)} \& \mathrm{id}_{Ca_{H_{12}}}) \\ &= \mathrm{id}_{\mathcal{I}_s(Ca_{H_{11}} \& c(G_1,i_1) \& Ca_{H_{12}})} = \mathrm{id}_{B_1} \end{aligned}$$

    where we have used fact 8.3.30,(4).

- $H_{11} \neq H_{21}$. We can wlog. assume that H_{11} is shorter than H_{21}. Then there exists H such that $H_{12} = H \& G_2 \& H_{22}$, $H_{21} = H_{11} \& G_1 \& H$. Now define

  $$\begin{aligned} t'_1 &= \mathcal{I}_s(\mathrm{id}_{Ca_{H_{11}}} \& \mathrm{id}_{c(G_1,i_1)} \& \mathrm{id}_{Ca_H} \& t(G_2,i_2) \& \mathrm{id}_{Ca_{H_{22}}}) \\ t'_2 &= \mathcal{I}_{s_1}(\mathrm{id}_{Ca_{H_{11}}} \& \mathcal{R}(\mathcal{I}_{s(G_1,i_1)}(t(G_1,i_1))) \& \mathrm{id}_{Ca_H} \& \mathrm{id}_{c(G_2,i_2)} \& \mathrm{id}_{Ca_{H_{22}}}) \end{aligned}$$

We can easily calculate

$$\begin{array}{rl} & t_1' \star t_2' \\ = & \mathcal{I}_{s_1}(\mathrm{id}_{Ca_{H_{11}}} \& \mathcal{R}(\mathcal{I}_{s(G_1,i_1)}(t(G_1,i_1))) \& \mathrm{id}_{Ca_H} \& t(G_2,i_2) \& \mathrm{id}_{Ca_{H_{22}}}) \\ = & t_1 \star t_2 \end{array}$$

To show that $1 \vdash_u t_1'$, notice that we can write $c(G_1, i_1)$ on the form $\mathcal{I}_{s'}(Ca_{H'})$ for some s',H'. To show that $1 \vdash_f t_2'$, apply the same observation to $c(G_2, i_2)$.

□

The normalization lemma

Lemma 8.4.15 Suppose $1 \vdash^* t$: $B \Rightarrow B'$, with B' not failure. Then there exists B'', t_1 and t_2 such that $1 \vdash_u^* t_1$: $B \Rightarrow B''$, $1 \vdash_f^* t_2$: $B'' \Rightarrow B'$, $t_1 \star t_2 = t$.

From transitions being non-increasing we conclude that B'' is not failure, and from t_1, t_2 and $t_1 \star t_2$ trivially being $\neq \perp$ we conclude $t \neq \perp$. Moreover, if B' is not empty neither is B''. □

Proof: There exists $t_1, \ldots, t_n$ such that $t = t_1 \star \ldots \star t_n$ and such that for each $i \in \{1 \ldots n\}$ either $1 \vdash_u t_i$ or $1 \vdash_f t_i$. We will use induction in the number of times an application of the fold-rule precedes (not necessarily immediately) an application of the unfold-rule. If this number is zero, we are through. Otherwise there exists i, $1 \leq i < n$, such that $1 \vdash_f t_i$, $1 \vdash_u t_{i+1}$. Now apply lemma 8.4.14. Two possibilities:

- $t_i \star t_{i+1} = \mathrm{id}_\_$. Then

  $$t = t_1 \star \ldots \star t_{i-1} \star t_{i+2} \star \ldots \star t_n$$

  with a strictly smaller number of "inversions".

- There exists t_i', t_{i+1}' such that $t_i' \star t_{i+1}' = t_i \star t_{i+1}$, and such that $1 \vdash_u t_i'$, $1 \vdash_f t_{i+1}'$. Now

  $$t = t_1 \star \ldots \star t_{i-1} \star t_i' \star t_{i+1}' \star t_{i+2} \star \ldots \star t_n$$

  and again the number of inversions has decreased.

□

8.4.7 Unfolding at level 2

Now assume that we have defined $\mathcal{R}_1$, a finite set of rules at level 1, such that $t \in \mathcal{R}_1$ implies that $1 \vdash^* t$.

We now define what it means for a transition t from B to B' to be a level 2 unfolding step, to be written $2 \vdash t\colon B \Rightarrow B'$:

$$\frac{t \in \mathcal{R}_1}{2 \vdash \mathcal{I}_s(\mathrm{id}_{Ca_{H_1}} \& t \& \mathrm{id}_{Ca_{H_2}})} \tag{8.20}$$

Next we define what it means for a transition t from B to B' to be a level 2 unfolding, to be written $2 \vdash^* t\colon B \Rightarrow B'$:

$$\frac{2 \vdash t\colon B \Rightarrow B'}{2 \vdash^* t\colon B \Rightarrow B'} \tag{8.21}$$

$$\frac{}{2 \vdash^* \mathrm{id}_B\colon B \Rightarrow B} \tag{8.22}$$

$$\frac{2 \vdash^* t\colon B \Rightarrow B', 2 \vdash^* t'\colon B' \Rightarrow B''}{2 \vdash^* t \star t'\colon B \Rightarrow B''} \tag{8.23}$$

Fact 8.4.16 Suppose $2 \vdash t$ or $2 \vdash^* t$. Then $1 \vdash^* t$. □

PROOF: For $2 \vdash t$, exploit e.g. fact 8.4.13. For $2 \vdash^* t$, a straight forward induction in the derivation tree. □

Definition 8.4.17 We say that B loops at level 2 *by some strategy* if for all $n \geq 1$ there exists a t_n and a B_n not failure such that $2 \vdash t_n\colon B_{n-1} \Rightarrow B_n$ (here $B_0 = B$). □

Definition 8.4.18 We say that t is a *fair level 2 step* if t is of form $\mathcal{I}_s(r_1 \& \ldots \& r_k)$, $k \geq 1$, where $r_1 \ldots r_k$ are level 1 rules.

For a fair level 2 step t, we clearly have $2 \vdash^* t$. □

Definition 8.4.19 We say that B loops at level 2 *by a fair strategy* if for all $n \geq 1$ there exists a fair level 2 step t_n and a B_n not failure such that $2 \vdash^* t_n\colon B_{n-1} \Rightarrow B_n$ (here $B_0 = B$). □

Definition 8.4.20 We say that t is a $\mathcal{LR}$ *level 2 step* if t is of form $\mathcal{I}_s(r \& \mathrm{id}\_)$ with r a level 1 rule. □

Definition 8.4.21 We say that B loops at level 2 *by a* $\mathcal{LR}$ *strategy* if for all $n \geq 1$ there exists a $\mathcal{LR}$ level 2 step t_n and a B_n not failure such that $2 \vdash t_n\colon B_{n-1} \Rightarrow B_n$ (here $B_0 = B$). □

8.5 Conditions for termination preservation

We are now in position to prove that condition 8.1.3, condition 8.1.1 and condition 8.1.2 indeed are sufficient for ensuring total correctness (wrt. the corresponding evaluation strategy).

Theorem 8.5.1 Assume all rules in $\mathcal{R}_1$ satisfy $\mathcal{A}(1)$. Then if B loops at level 2 by some strategy, it also loops at level 1 by some strategy. $\square$

PROOF: Let, for all $n \geq 1$, be given t_n and B_n not failure such that $2 \vdash t_n\colon B_{n-1} \Rightarrow B_n$. Define $t'_n = t_1 \star \ldots \star t_n$. Now $2 \vdash^* t'_n\colon B \Rightarrow B_n$, and by fact 8.4.16 also $1 \vdash^* t'_n\colon B \Rightarrow B_n$. By lemma 8.4.15, there exists t''_n, t'''_n and B'_n such that $1 \vdash^*_u t''_n\colon B \Rightarrow B'_n$, $1 \vdash^*_f t'''_n\colon B'_n \Rightarrow B_n$, $t'_n = t''_n \star t'''_n$ and B'_n not failure.

Due to the assumption of the theorem, each t_i will satisfy $\mathcal{A}(1)$. Then, by lemma 8.3.34, each t'_n will satisfy $\mathcal{A}(n)$. But then also t''_n will satisfy $\mathcal{A}(n)$. By lemma 8.4.5, this shows that B loops at level 1 by some strategy. $\square$

Theorem 8.5.2 Assume all rules in $\mathcal{R}_1$ satisfy $\mathcal{F}(1)$. Then if B loops at level 2 by a fair strategy, it also loops at level 1 by a fair strategy. $\square$

PROOF: Let, for all $n \geq 1$, be given fair level 2 step t_n and B_n not failure nor empty such that $2 \vdash^* t_n\colon B_{n-1} \Rightarrow B_n$. Define $t'_n = t_1 \star \ldots \star t_n$. Now $2 \vdash^* t'_n\colon B \Rightarrow B_n$, and by fact 8.4.16 also $1 \vdash^* t'_n\colon B \Rightarrow B_n$. By lemma 8.4.15, there exists t''_n, t'''_n and B'_n such that $1 \vdash^*_u t''_n\colon B \Rightarrow B'_n$, $1 \vdash^*_f t'''_n\colon B'_n \Rightarrow B_n$, $t'_n = t''_n \star t'''_n$ and B'_n not failure nor empty.

Due to the assumption of the theorem, each t_i will satisfy $\mathcal{F}(1)$. Then, by lemma 8.3.32, each t'_n will satisfy $\mathcal{F}(n)$. But then also t''_n will satisfy $\mathcal{F}(n)$. By lemma 8.4.8, this shows that B loops at level 1 by a fair strategy. $\square$

Theorem 8.5.3 Assume all rules in $\mathcal{R}_1$ satisfy $\mathcal{L}(1)$. Then if B loops at level 2 by a $\mathcal{LR}$ strategy, it also loops at level 1 by a $\mathcal{LR}$ strategy. $\square$

PROOF: Let, for all $n \geq 1$, be given $\mathcal{LR}$ level 2 step t_n and B_n not failure such that $2 \vdash t_n\colon B_{n-1} \Rightarrow B_n$. Define $t'_n = t_1 \star \ldots \star t_n$. Now $2 \vdash^* t'_n\colon B \Rightarrow B_n$, and by fact 8.4.16 also $1 \vdash^* t'_n\colon B \Rightarrow B_n$. By lemma

8.4.15, there exists t''_n, t'''_n and B'_n such that $1 \vdash^*_u t''_n\colon B \Rightarrow B'_n$, $1 \vdash^*_f t'''_n\colon B'_n \Rightarrow B_n$, $t'_n = t''_n \star t'''_n$ and B'_n not failure.

Due to the assumption of the theorem, each t_i will satisfy $\mathcal{L}(1)$. Then, by lemma 8.3.37, we will have an increasing sequence

$$t'_1 \prec t'_2 \prec t'_3 \prec \dots$$

Now two possibilities:

- $E(t'_n)$ is not bounded. Then neither $E(t''_n)$ is bounded, so by lemma 8.4.11 B loops at level 1 by a $\mathcal{LR}$ strategy.

- $E(t'_n)$ is bounded. Then $L(t'_n)$ is unbounded, so also $L(t''_n)$ is unbounded. Then again lemma 8.4.11 tells us that B loops at level 1 by a $\mathcal{LR}$ strategy.

$\Box$

8.6 Working with the full search tree

The development in this section has been motivated in section 8.1.5.

Definition 8.6.1 A *configuration* C (over K) is a family of basic configurations over K, i.e. consists of an index set I and a mapping B which to each $i \in I$ assigns a basic configuration over K. $\Box$

Two new operators will be defined, $+$ and $\mathsf{P}(\_)$:

Definition 8.6.2 Given configurations $C_1 = (I_1, B_1)$ and $C_2 = (I_2, B_2)$, over the same K. Now we define $C_1{+}C_2 = (I, B)$ by letting $I = I_1{+}I_2$ (where $+$ denotes disjoint union); and by letting $B(\mathsf{in}_1(i)) = B_1(i)$, $B(\mathsf{in}_2(i)) = B_2(i)$. $\Box$

Definition 8.6.3 Given configuration $C = (I, B)$. Let

$$I' = \{i \in I | B(i) \text{ is not failure}\}$$

Now we define $\mathsf{P}(C) = (I', B')$ where $B'(i') = B(i')$. We say that C is *pruned* if $C = \mathsf{P}(C)$. $\Box$

& will be extended to configurations such that & distributes over $+$. That is, if $C = (I, B)$ and $C' = (I', B')$ then $C\&C' = (I \times I', B'')$ where $B''(i, i') = B(i)\&B'(i')$. Notice that this is possible only because configurations are multisets with $+$ as multiset union; if configurations had been sequences with $+$ as concatenation it would be impossible to make & left-distributive as well as right-distributive.

$\mathcal{I}_{\_}(\_)$ will be extended to configurations such that

$$\mathcal{I}_s(C_1{+}C_2) = \mathcal{I}_s(C_1){+}\mathcal{I}_s(C_2)$$

A lot of new identities hold, not to be stated explicitly here – most are very trivial.

8.6.1 Transitions

Definition 8.6.4 A transition t from $C = (I, B)$ to $C' = (I', B')$ now to each $(i, i') \in I \times I'$ assigns a set $t(i, i')$ of U-mirrors from $B(i)$ to $B'(i')$. Moreover, we will demand t to be non-increasing: if $t(i, i')$ is non-empty, $(B(i), B'(i'))$ must be a non-increasing pair. $\square$

Definition 8.6.5 Given transition t from $C = (I, B)$ to $C' = (I', B')$ and transition t' from C' to $C'' = (I'', B'')$. $t{\star}t'$, a transition from C to C'', is now defined as follows:

$$(t{\star}t')(i, i'') = \{m{\star}m' | \exists i' \in I' : m \in t(i, i'), m' \in t(i', i'')\}$$

$\square$

& on transitions is defined in a similar vein:

Definition 8.6.6 Given transition t_1 from $C_1 = (I_1, B_1)$ to $C_1' = (I_1', B_1')$, and given transition t_2 from $C_2 = (I_2, B_2)$ to $C_2' = (I_2', B_2')$. Then $t_1\&t_2$, a transition from $C_1\&C_2$ to $C_1'\&C_2'$, is defined by

$$(t_1\&t_2)((i_1, i_2), (i_1', i_2')) = \{m_1\&m_2 | m_1 \in t_1(i_1, i_1'), m_2 \in t_2(i_2, i_2')\}$$

$\square$

Definition 8.6.7 Given transition t from $C = (I, B)$ to $C' = (I', B')$. Then $\mathcal{I}_s(t)$, a transition from $\mathcal{I}_s(C)$ to $\mathcal{I}_s(C')$, is given by

$$\mathcal{I}_s(t)(i, i') = t(i, i')$$

$\square$

Definition 8.6.8 Given configuration $C = (I, B)$, we define id_C by

$$\mathrm{id}_C(i, i) = \{\mathrm{id}_{B(i)}\}, \mathrm{id}_C(i, i') = \emptyset \text{ for } i \neq i'$$

□

Definition 8.6.9 Given transition t_1 from $C_1 = (I_1, B_1)$ to $C_1' = (I_1', B_1')$, and given transition t_2 from $C_2 = (I_2, B_2)$ to $C_2' = (I_2', B_2')$ (suppose C_1 and C_2 configurations over the same K). Then $t_1{+}t_2$, a transition from $C_1{+}C_2$ to $C_1'{+}C_2'$, is defined by

$$\begin{array}{rcl}
(t_1{+}t_2)(\mathsf{in}_1(i_1), \mathsf{in}_1(i_1')) & = & t_1(i_1, i_1') \\
(t_1{+}t_2)(\mathsf{in}_2(i_2), \mathsf{in}_2(i_2')) & = & t_2(i_2, i_2') \\
(t_1{+}t_2)(\mathsf{in}_1(i_1), \mathsf{in}_2(i_2')) & = & \emptyset \\
(t_1{+}t_2)(\mathsf{in}_2(i_2), \mathsf{in}_1(i_1')) & = & \emptyset
\end{array}$$

□

Definition 8.6.10 Given transition t from $C_1 = (I_1, B_1)$ to $C_2 = (I_2, B_2)$. Let $\mathsf{P}(C_1) = (I_1', B_1')$, $\mathsf{P}(C_2) = (I_2', B_2')$. Now define $\mathsf{P}(t)$, a transition from $\mathsf{P}(C_1)$ to $\mathsf{P}(C_2)$, by

$$\mathsf{P}(t)(i_1', i_2') = t(i_1', i_2')$$

□

Definition 8.6.11 Given transition t from $C = (I, B)$ to $C' = (I', B')$, we say that t is reversible iff the following holds for all (i, i') with $t(i, i') \neq \emptyset$: $(B'(i'), B(i))$ is a non-increasing pair, and for all $m \in t(i, i')$ m is a working U-mirror.

If t is reversible we can define $\mathcal{R}(t)$, a transition from C' to C, by stipulating

$$\mathcal{R}(t)(i', i) = \{\mathcal{R}(m) | m \in t(i, i')\}$$

□

Again, a lot of algebraic identities hold – most are quite trivial. Let us just show that

$$(t_1 \& t_2) \star (t_1' \& t_2') = (t_1 \star t_1') \& (t_2 \star t_2') \tag{8.24}$$

where t_1 is a transition from $C_1 = (I_1, B_1)$ to $C_1' = (I_1', B_1')$, t_1' is a transition from C_1' to $C_1'' = (I_1'', B_1'')$, t_2 is a transition from $C_2 = (I_2, B_2)$

to $C_2' = (I_2', B_2')$ and t_2' is a transition from C_2' to $C_2'' = (I_2'', B_2'')$. Now the left hand side of (8.24) as well as the right hand side will be a transition from $C_1 \& C_2$ to $C_1'' \& C_2''$. And for $i_1 \in I_1$, $i_2 \in I_2$, $i_1'' \in I_1''$ and $i_2'' \in I_2''$ we have

$$\begin{array}{rl} & ((t_1 \& t_2) \star (t_1' \& t_2'))((i_1, i_2), (i_1'', i_2'')) \\ = & \{m'' | \exists (i_1', i_2'), \exists m \in (t_1 \& t_2)((i_1, i_2), (i_1', i_2')), \\ & \exists m' \in (t_1' \& t_2')((i_1', i_2'), (i_1'', i_2'')) : m'' = m \star m'\} \\ = & \{m'' | \exists (i_1', i_2'), \exists m_1 \in t_1(i_1, i_1'), \exists m_2 \in t_2(i_2, i_2'), \\ & \exists m_1' \in t_1'(i_1', i_1''), \exists m_2' \in t_2'(i_2', i_2'') : m'' = (m_1 \& m_2) \star (m_1' \& m_2')\} \\ = & \{m'' | \exists (i_1', i_2'), \exists m_1 \in t_1(i_1, i_1'), \exists m_2 \in t_2(i_2, i_2'), \\ & \exists m_1' \in t_1'(i_1', i_1''), \exists m_2' \in t_2'(i_2', i_2'') : m'' = (m_1 \star m_1') \& (m_2 \star m_2')\} \\ = & ((t_1 \star t_1') \& (t_2 \star t_2'))((i_1, i_2), (i_1'', i_2'')) \end{array}$$

where we have used fact 8.3.30, (2).

Also we have that

$$\mathsf{P}(t_1 \star t_2) = \mathsf{P}(t_1) \star \mathsf{P}(t_2)$$

This holds only because we demand transitions to be non-increasing.

8.6.2 The level 0 rules

For each $G \in U$, there exists a rule $t(G) \in \mathcal{R}_0$ from Ca_G to $c(G)$, where $c(G) = \{c(G, i) | i \in OI(G)\}$. Here $m \in t(G)(i)$ iff $m \in t(G, i)$.

8.6.3 Unfolding at level 1

We now define what it means for a transition t from C to C' to be a *level 1c unfolding step* [14], to be written $1\ \mathrm{c}\vdash_u t$: $C \Rightarrow C'$.

$$\frac{t(G) \in \mathcal{R}_0}{1\ \mathrm{c}\vdash_u \mathcal{I}_s(\mathrm{id}_{Ca_{H_1}} \& t(G) \& \mathrm{id}_{Ca_{H_2}})} \tag{8.25}$$

$$\frac{}{1\ \mathrm{c}\vdash_u \mathrm{id}_C} \tag{8.26}$$

$$\frac{1\ \mathrm{c}\vdash_u t_1, 1\ \mathrm{c}\vdash_u t_2}{1\ \mathrm{c}\vdash_u t_1 + t_2} \tag{8.27}$$

[14]The "c" to denote that the *c*omplete search tree is modeled.

Fact 8.6.12 If $1 \;\mathsf{c}\!\vdash_u t$, then $1 \;\mathsf{c}\!\vdash_u t\&\mathrm{id}\_$ and $1 \;\mathsf{c}\!\vdash_u \mathrm{id}\_\&t$. Also, $1 \;\mathsf{c}\!\vdash_u \mathcal{I}_s(t)$. □

Next we define what it means for a transition t from C to C' to be a level 1c unfolding, to be written $1 \;\mathsf{c}\!\vdash_u^* t\colon C \Rightarrow C'$:

$$\frac{1 \;\mathsf{c}\!\vdash_u t\colon C \Rightarrow C'}{1 \;\mathsf{c}\!\vdash_u^* \mathsf{P}(t)\colon \mathsf{P}(C) \Rightarrow \mathsf{P}(C')} \tag{8.28}$$

$$\frac{1 \;\mathsf{c}\!\vdash_u^* t\colon C \Rightarrow C', 1 \;\mathsf{c}\!\vdash_u^* t'\colon C' \Rightarrow C''}{1 \;\mathsf{c}\!\vdash_u^* t\star t'\colon C \Rightarrow C''} \tag{8.29}$$

Observe that if $1 \;\mathsf{c}\!\vdash_u^* t\colon C \Rightarrow C'$, then $t = \mathsf{P}(t)$, $C = \mathsf{P}(C)$ and $C' = \mathsf{P}(C')$.

Fact 8.6.13 If $1 \;\mathsf{c}\!\vdash_u^* t_1$ and $1 \;\mathsf{c}\!\vdash_u^* t_2$, then $1 \;\mathsf{c}\!\vdash_u^* t_1\&t_2$, $1 \;\mathsf{c}\!\vdash_u^* t_1{+}t_2$ and $1 \;\mathsf{c}\!\vdash_u^* \mathsf{P}(\mathcal{I}_s(t_1))$. □

PROOF: Inductions in the derivation tree:

- Concerning &, it will be enough to show that
  $1 \;\mathsf{c}\!\vdash_u^* t$ implies $1 \;\mathsf{c}\!\vdash_u^* t\&\mathrm{id}_C$ (and $1 \;\mathsf{c}\!\vdash_u^* \mathrm{id}_C\&t$) for C with $\mathsf{P}(C) = C$, as

  $$t_1\&t_2 = (t_1\star\mathrm{id}\_)\&(\mathrm{id}\_\star t_2) = (t_1\&\mathrm{id}\_)\star(\mathrm{id}\_\&t_2)$$

  If $t = t_1\star t_2$ with $1 \;\mathsf{c}\!\vdash_u^* t_1$, $1 \;\mathsf{c}\!\vdash_u^* t_2$ this follows from

  $$t\&\mathrm{id}\_ = (t_1\&\mathrm{id}\_)\star(t_2\&\mathrm{id}\_)$$

  If $t = \mathsf{P}(t')$ with $1 \;\mathsf{c}\!\vdash_u t'$, first note that by fact 8.6.12 $1 \;\mathsf{c}\!\vdash_u t'\&\mathrm{id}\_$. Thus $1 \;\mathsf{c}\!\vdash_u^* \mathsf{P}(t'\&\mathrm{id}\_)$, i.e. $1 \;\mathsf{c}\!\vdash_u^* t\&\mathrm{id}\_$.

- Concerning +, it will be enough to show that $1 \;\mathsf{c}\!\vdash_u^* t$ implies
  $1 \;\mathsf{c}\!\vdash_u^* t{+}\mathrm{id}_C$ with C such that $C = \mathsf{P}(C)$ – as

  $$t_1{+}t_2 = (t_1{+}\mathrm{id}\_)\star(\mathrm{id}\_{+}t_2)$$

  If $t = t_1\star t_2$, this follows from

  $$t{+}\mathrm{id}\_ = (t_1{+}\mathrm{id}\_)\star(t_2{+}\mathrm{id}\_)$$

  If $t = \mathsf{P}(t')$ with $1 \;\mathsf{c}\!\vdash_u t'$, first note that $1 \;\mathsf{c}\!\vdash_u t'{+}\mathrm{id}\_$.
  Thus $1 \;\mathsf{c}\!\vdash_u^* \mathsf{P}(t'{+}\mathrm{id}\_)$, i.e. $1 \;\mathsf{c}\!\vdash_u^* t{+}\mathrm{id}\_$.

- Concerning $\mathcal{I}_{\_}(\_)$, first suppose $1 \; \mathrm{c}\!\vdash_u^* t$ because $t = \mathsf{P}(t')$, $1 \; \mathrm{c}\!\vdash_u t'$. Now also $1 \; \mathrm{c}\!\vdash_u \mathcal{I}_s(t')$, and thus $1 \; \mathrm{c}\!\vdash_u^* \mathsf{P}(\mathcal{I}_s(t'))$, i.e. also $1 \; \mathrm{c}\!\vdash_u^* \mathsf{P}(\mathcal{I}_s(t))$.

  Next suppose $t = t_1 \star t_2$. By induction, $1 \; \mathrm{c}\!\vdash_u^* \mathsf{P}(\mathcal{I}_s(t_1))$ and $1 \; \mathrm{c}\!\vdash_u^* \mathsf{P}(\mathcal{I}_s(t_2))$. The claim now follows from

  $$\mathsf{P}(\mathcal{I}_s(t_1)) \star \mathsf{P}(\mathcal{I}_s(t_2)) = \mathsf{P}(\mathcal{I}_s(t_1 \star t_2))$$

□

If $1 \; \mathrm{c}\!\vdash_u^* t$, there exists $t_1 \ldots t_n$ such that $t = \mathsf{P}(t_1) \star \ldots \star \mathsf{P}(t_n)$, with $1 \; \mathrm{c}\!\vdash_u t_i$ for $i = 1 \ldots n$. Again, we can define the length of a transition t as the minimal n which can be used ($n \geq 1$).

Observation 8.6.14 Suppose $1 \; \mathrm{c}\!\vdash_u^* t$: $B \Rightarrow C$, with $C = (I, B)$. Then for all $i \in I$ there exists t_i such that $1 \vdash_u^* t_i$: $B \Rightarrow B(i)$. We say that $t_i = \pi_i(t)$. □

The diamond lemma, revisited

Lemma 8.6.15 Suppose $1 \; \mathrm{c}\!\vdash_u t_1$: $C \Rightarrow C_1$ and $1 \; \mathrm{c}\!\vdash_u t_2$: $C \Rightarrow C_2$. Then there exists C', transition t'_1 with $1 \; \mathrm{c}\!\vdash_u t'_1$: $C_1 \Rightarrow C'$ and transition t'_2 with $1 \; \mathrm{c}\!\vdash_u t'_2$: $C_2 \Rightarrow C'$ such that $t_1 \star t'_1 = t_2 \star t'_2$. □

Proof: Much as the proof of lemma 8.4.2. A brief sketch: we can assume C to consist of a single basic configuration (if C is the union of several basic configurations each of these can be "treated" separately).

If $t_1 = \mathrm{id}_C$, choose $t'_1 = t_2$, $t'_2 = \mathrm{id}_{C_2}$. Similarly if $t_2 = \mathrm{id}_C$. So in the following we can assume that t_1 as well as t_2 are derived by means of (8.25). If "the same G" is unfolded, we take $t'_1 = t'_2 = \mathrm{id}_{C_1}$. Otherwise, we can – dispensing with the id_{Ca_H}-parts – write $t_1 = \mathcal{I}_s(\mathrm{id}_{Ca_{G_1}} \& t(G_2))$, $t_2 = \mathcal{I}_s(t(G_1) \& \mathrm{id}_{Ca_{G_2}})$. Then C_1 and C_2 take the form

$$\begin{aligned} C_1 &= \{\mathcal{I}_s(Ca_{G_1} \& c(G_2, i)) | i \in OI(G_2)\} \\ C_2 &= \{\mathcal{I}_s(c(G_1, i) \& Ca_{G_2}) | i \in OI(G_1)\} \end{aligned}$$

Now define t'_1, t'_2 as follows:

$$\begin{aligned} t'_1 &= \mathcal{I}_s(t(G_1) \& \mathrm{id}_{c(G_2)}) \\ t'_2 &= \mathcal{I}_s(\mathrm{id}_{c(G_1)} \& t(G_2)) \end{aligned}$$

That e.g. $1 \; \mathrm{c}\!\vdash_u t'_1$ follows from the fact that t'_1 can be written on the form

$$t'_1 = \sum_{i \in OI(G_2)} \mathcal{I}_s(t(G_1) \& \mathrm{id}_{c(G_2, i)})$$

where again $c(G_2, i)$ can be written on the form $\mathcal{I}_{s'}(Ca_{H'})$.

That $t_1 \star t'_1 = t_2 \star t'_2$ follows from the fact that both equal $\mathcal{I}_s(t(G_1)\&t(G_2))$.
□

The Church-Rosser lemma, revisited

Lemma 8.6.16 Suppose 1 $\mathrm{c}\vdash^*_u t_1$: $C \Rightarrow C_1$ and 1 $\mathrm{c}\vdash^*_u t_2$: $C \Rightarrow C_2$. Then there exists C_3 and transitions t_3, t_4 such that 1 $\mathrm{c}\vdash^*_u t_3$: $C_1 \Rightarrow C_3$, 1 $\mathrm{c}\vdash^*_u t_4$: $C_2 \Rightarrow C_3$, and such that $t_1 \star t_3 = t_2 \star t_4$.

Moreover, the length of t_3 is less than or equal the length of t_2, and the length of t_4 is less than or equal the length of t_1. □

Proof: We use induction in the length of t_1 plus the length of t_2. For the induction step, proceed as in the proof of lemma 8.4.3. So assume that both transitions have length 1. The situation is as follows: we have 1 $\mathrm{c}\vdash_u t'_1$: $C' \Rightarrow C'_1$ and 1 $\mathrm{c}\vdash_u t'_2$: $C'' \Rightarrow C'_2$, with $t_1 = \mathsf{P}(t'_1)$, $t_2 = \mathsf{P}(t'_2)$, and $C = \mathsf{P}(C') = \mathsf{P}(C'')$. We can assume that $C' = C''$, as we can "expand" t'_1 and t'_2. Now apply lemma 8.6.15, to find t'_3, t'_4 and C'_3 with 1 $\mathrm{c}\vdash_u t'_3$: $C'_1 \Rightarrow C'_3$, 1 $\mathrm{c}\vdash_u t'_4$: $C'_2 \Rightarrow C'_3$, $t'_1 \star t'_3 = t'_2 \star t'_4$. Then define $C_3 = \mathsf{P}(C'_3)$, $t_3 = \mathsf{P}(t'_3)$, $t_4 = \mathsf{P}(t'_4)$. Then 1 $\mathrm{c}\vdash^*_u t_3$, 1 $\mathrm{c}\vdash^*_u t_4$, and

$$t_1 \star t_3 = \mathsf{P}(t'_1) \star \mathsf{P}(t'_3) = \mathsf{P}(t'_1 \star t'_3) = \mathsf{P}(t'_2 \star t'_4) = t_2 \star t_4$$

□

8.6.4 Level 1 semantics

We say that a configuration $C = (I, B)$ is in *normal form* iff C is pruned and for all $i \in I$ $B(i)$ is empty.

Lemma 8.6.16, together with the observation that if C is in normal form and 1 $\mathrm{c}\vdash^*_u t$: $C \Rightarrow C'$ then $C = C'$, shows that the following is well-defined:

Definition 8.6.17 Given basic configuration B (not failure). Suppose 1 $\mathrm{c}\vdash^*_u t$: $B \Rightarrow C$ with C in normal form. Then $[\![\mathrm{B}]\!]_1 = C$.

If no such t and C exists, $[\![\mathrm{B}]\!]_1 = \perp$. □

Fact 8.6.18 $[\![\mathrm{B}]\!]_1 = \perp$ iff B loops at level 1 by a fair strategy (as defined in definition 8.4.7) □

PROOF: Suppose $[\![\mathrm{B}]\!]_1 = \perp$. Given $n \geq 0$. It is easily seen that there will exist t_n and $C' = (I', B')$ such that $1 \;\mathrm{c}\!\vdash_u^* t_n$: $B \Rightarrow C'$, and such that for all $i \in I'$ we have $1 \vdash_u^* \pi_i(t_n)$: $B \Rightarrow B'(i)$ where either $B'(i)$ is empty or $\pi_i(t_n)$ is composed of n fair level 1 unfolding steps. Now, there exists at least one $i \in I'$ where $B'(i)$ is not empty (otherwise C' would be in normal form). Hence we conclude that B loops at level 1 by a fair strategy.

Conversely, suppose there exists C in normal form such that $1 \;\mathrm{c}\!\vdash_u^* t$: $B \Rightarrow C$. Then there exists U-forests $f_1 \ldots f_k$ such that $\mathcal{U}_{f_i}(B)$ is either empty or failure for all i. Moreover, there exists a n such that for all U-forests f where the shortest working path is longer than n, there exists an i such that f can be written as $f_i \star f'$ for some f'. Hence for such f also $\mathcal{U}_f(B)$ is empty or failure, showing that B does not loop at level 1 by a fair strategy. □

$\mathcal{LR}$ semantics

We say that t is a $\mathcal{LR}$ level 1c single step if t takes the form $t = \mathcal{I}_s(t(G)\&\mathrm{id}_B)$, with $t(G) \in \mathcal{R}_0$. We say that t is a $\mathcal{LR}$ level 1c step if t takes the form $t = t_1 + \ldots + t_k$, at least one t_i being a $\mathcal{LR}$ level 1c single step and the rest being of form id_B, B empty. We say that t is a $\mathcal{LR}$ level 1c unfolding if t takes the form $t = \mathsf{P}(t_1) \star \ldots \star \mathsf{P}(t_i)$, each t_i being a $\mathcal{LR}$ level 1c step.

Definition 8.6.19 Given basic configuration B. Suppose $1 \;\mathrm{c}\!\vdash_u^* t$: $B \Rightarrow C$ with C in normal form, where t is a $\mathcal{LR}$ level 1c unfolding. Then $[\![\mathrm{B}]\!]_1^L = C$.

If no such t and C exists, $[\![\mathrm{B}]\!]_1^L = \perp$. □

Fact 8.6.20 $[\![\mathrm{B}]\!]_1^L = \perp$ iff B loops at level 1 by a $\mathcal{LR}$ strategy (as defined in definition 8.4.10). □

PROOF: Suppose $[\![\mathrm{B}]\!]_1^L = \perp$. Given $n \geq 0$. It is easily seen that there will exist t_n and $C' = (I', B')$ such that $1 \;\mathrm{c}\!\vdash_u^* t_n$: $B \Rightarrow C'$, and such that for all $i \in I'$ we have $1 \vdash_u^* \pi_i(t_n)$: $B \Rightarrow B'(i)$ where either B_i' is empty or $\pi_i(t_n)$ is composed of n $\mathcal{LR}$ level 1 unfolding steps. Now, there exists at least one $i \in I'$ where $B'(i)$ is not empty (otherwise C' would be in normal form). Hence we conclude that B loops at level 1 by a $\mathcal{LR}$ strategy.

Conversely, suppose there exists C in normal form such that $1\ \mathrm{c}\!\vdash^*_u t\colon B \Rightarrow C$, with t a $\mathcal{LR}$ level 1c unfolding. Then there exists U-forests $f_1 \ldots f_k$ such that $\mathcal{U}_{f_i}(B)$ is either empty or failure for all i. Moreover, there exists a n such that for all $\mathcal{LR}$ U-forests f of size greater than n, there exists an i such that f can be written as $f_i{\star}f'$ for some f'. Hence for such f also $\mathcal{U}_f(B)$ is empty or failure, showing that B does not loop at level 1 by a $\mathcal{LR}$ strategy. □

8.6.5 Folding at level 1

We now define what it means for a transition t from C to C' to be a level 1c folding step, to be written $1\ \mathrm{c}\!\vdash_f t\colon C \Rightarrow C'$.

We will assume the existence of a partial (perhaps multivalued) function $s(G)$ such that $\mathsf{P}(\mathcal{I}_{s(G)}(t(G)))$ is reversible.

$$\frac{t(G) \in \mathcal{R}_0}{1\ \mathrm{c}\!\vdash_f \mathcal{I}_s(\mathrm{id}_{Ca_{H_1}} \& \mathcal{R}(\mathsf{P}(\mathcal{I}_{s(G)}(t(G)))) \& \mathrm{id}_{Ca_{H_2}})} \tag{8.30}$$

$$\frac{}{1\ \mathrm{c}\!\vdash_f \mathrm{id}_C} \tag{8.31}$$

$$\frac{1\ \mathrm{c}\!\vdash_f t_1, 1\ \mathrm{c}\!\vdash_f t_2}{1\ \mathrm{c}\!\vdash_f t_1{+}t_2} \tag{8.32}$$

Next we define what it means for a transition t from C to C' to be a level 1c folding, to be written $1\ \mathrm{c}\!\vdash^*_f t\colon C \Rightarrow C'$:

$$\frac{1\ \mathrm{c}\!\vdash_f t\colon C \Rightarrow C'}{1\ \mathrm{c}\!\vdash^*_f \mathsf{P}(t)\colon \mathsf{P}(C) \Rightarrow \mathsf{P}(C')} \tag{8.33}$$

$$\frac{1\ \mathrm{c}\!\vdash^*_f t\colon C \Rightarrow C', 1\ \mathrm{c}\!\vdash^*_f t'\colon C' \Rightarrow C''}{1\ \mathrm{c}\!\vdash^*_f t{\star}t'\colon C \Rightarrow C''} \tag{8.34}$$

8.6.6 Unfold/fold at level 1

We now define what it means for a transition t from C to C' to be a level 1c transition, to be written $1\ \mathrm{c}\!\vdash^* t\colon C \Rightarrow C'$.

$$\frac{1\ \mathrm{c}\!\vdash_u t\colon C \Rightarrow C'}{1\ \mathrm{c}\!\vdash^* \mathsf{P}(t)\colon \mathsf{P}(C) \Rightarrow \mathsf{P}(C')} \tag{8.35}$$

$$\frac{1 \text{ c}\vdash_f t\colon C \Rightarrow C'}{1 \text{ c}\vdash^* \mathsf{P}(t)\colon \mathsf{P}(C) \Rightarrow \mathsf{P}(C')} \tag{8.36}$$

$$\frac{1 \text{ c}\vdash^* t\colon C \Rightarrow C', 1 \text{ c}\vdash^* t'\colon C' \Rightarrow C''}{1 \text{ c}\vdash^* t\star t'\colon C \Rightarrow C''} \tag{8.37}$$

Fact 8.6.21 If 1 c$\vdash^*$ t_1 and 1 c$\vdash^*$ t_2, then also 1 c$\vdash^*$ $t_1\&t_2$ and 1 c$\vdash^*$ $\mathsf{P}(\mathcal{I}_s(t_1))$. □

The switching lemma, revisited

Lemma 8.6.22 Suppose 1 c$\vdash^*$ t_1: $C_1 \Rightarrow C$ is a folding step, i.e. is derived by means of rule (8.36), and suppose 1 c$\vdash^*$ t_2: $C \Rightarrow C_2$ is an unfolding step, i.e. is derived by means of rule (8.35). Then there exists t_3, t_4 and C_3 such that 1 c$\vdash^*$ t_3: $C_1 \Rightarrow C_3$ by an unfolding step; 1 c$\vdash^*$ t_4: $C_3 \Rightarrow C_2$ by a folding step; and $t_1\star t_2 = t_3\star t_4$. □

Proof: (A sketch only) There exists t_1', t_2', C_1', C_2', C' and C'' such that 1 c$\vdash_f$ t_1': $C_1' \Rightarrow C'$, 1 c$\vdash_u$ t_2': $C'' \Rightarrow C_2'$, $t_1 = \mathsf{P}(t_1')$, $t_2 = \mathsf{P}(t_2')$ and $C = \mathsf{P}(C') = \mathsf{P}(C'')$. It is not hard to see that we can assume that t_1' and t_2' are derived by means of (8.30) and (8.25) respectively, and that we can assume C to be a singleton (i.e. not $\emptyset$).

There are two cases (where we dispense with writing the id_{Ca_H}-parts):

1. t_1', t_2' takes the form

$$\begin{aligned} t_1' &= \mathcal{I}_{s_1}(\mathcal{R}(\mathsf{P}(\mathcal{I}_{s(G)}(t(G))))) \\ t_2' &= \mathcal{I}_{s_2}(t(G)) \end{aligned}$$

   Thus $t_1 = \mathcal{R}(\mathsf{P}(\mathcal{I}_{s_1\star s(G)}(t(G))))$, $t_2 = \mathsf{P}(\mathcal{I}_{s_2}(t(G)))$. Then

$$\mathcal{I}_{s_1\star s(G)}(Ca_G) = C = \mathcal{I}_{s_2}(Ca_G)$$

   enabling us to conclude that $s_1\star s(G) = s_2$ and hence also $C_1 = C_2$. Now we can use $t_3 = t_4 = \mathrm{id}_{C_1}$. That $t_1\star t_2 = \mathrm{id}_{C_1}$ is an easy consequence of fact 8.3.30,(4).

2. t_1', t_2' takes the form

$$\begin{aligned} t_1' &= \mathcal{I}_{s_1}(\mathcal{R}(\mathsf{P}(\mathcal{I}_{s(G_1)}(t(G_1))))\&\mathrm{id}_{Ca_{G_2}}) \\ t_2' &= \mathcal{I}_{s_2}(\mathrm{id}_{Ca_{G_1}}\&t(G_2)) \end{aligned}$$

Now apply the usual technique: we infer that $s_1\star(s(G_1)\&\mathrm{id}_\_) = s_2$, and define

$$\begin{aligned} t'_4 &= \mathcal{I}_{s_1}(\mathcal{R}(\mathsf{P}(\mathcal{I}_{s(G_1)}(t(G_1))))\&\mathrm{id}_{c(G_2)}) \\ t'_3 &= \mathcal{I}_{s_2}(\mathrm{id}_{c(G_1)}\&t(G_2)) \end{aligned}$$

We now define $t_3 = \mathsf{P}(t'_3)$, $t_4 = \mathsf{P}(t'_4)$. Clearly $1 \;\mathrm{c}\!\vdash_u t'_3$, $1 \;\mathrm{c}\!\vdash_f t'_4$, and

$$t_3\star t_4 = \mathsf{P}(t'_3\star t'_4) = \mathsf{P}(t'_1\star t'_2) = t_1\star t_2$$

□

The normalization lemma, revisited

Lemma 8.6.23 Suppose $1 \;\mathrm{c}\!\vdash^* t\colon C \Rightarrow C'$. Then there exists t_1, t_2, C'' such that $1 \;\mathrm{c}\!\vdash^*_u t_1\colon C \Rightarrow C''$, $1 \;\mathrm{c}\!\vdash^*_f t_2\colon C'' \Rightarrow C'$, $t = t_1\star t_2$. □

PROOF: As the proof of lemma 8.4.15, now exploiting lemma 8.6.22. □

8.6.7 Unfolding at level 2

Now assume that we have defined $\mathcal{R}_1$, a finite set of rules at level 1. Assume that there is a bijective correspondence between $\mathcal{R}_1$ and U, such that the rule corresponding to G is a transition from Ca_G. Then there is no risk of a configuration being stuck (i.e. not in normal form but cannot be unfolded further).

We now define what it means for a transition t from C to C' to be a level 2c unfolding step, to be written $2 \;\mathrm{c}\!\vdash t\colon C \Rightarrow C'$.

$$\frac{t \in \mathcal{R}_1}{2 \;\mathrm{c}\!\vdash \mathcal{I}_s(\mathrm{id}_{Ca_{H_1}}\&t\&\mathrm{id}_{Ca_{H_2}})} \tag{8.38}$$

$$\frac{}{2 \;\mathrm{c}\!\vdash \mathrm{id}_C} \tag{8.39}$$

$$\frac{2 \;\mathrm{c}\!\vdash t_1, 2 \;\mathrm{c}\!\vdash t_2}{2 \;\mathrm{c}\!\vdash t_1 + t_2} \tag{8.40}$$

Next we define what it means for a transition t from C to C' to be a level 2c unfolding, to be written 2 $c\vdash^* t$: $C \Rightarrow C'$:

$$\frac{2\ c\vdash\ t\colon\ C \Rightarrow C'}{2\ c\vdash^*\ \mathsf{P}(t)\colon\ \mathsf{P}(C) \Rightarrow \mathsf{P}(C')} \tag{8.41}$$

$$\frac{2\ c\vdash^*\ t\colon\ C \Rightarrow C', 2\ c\vdash^*\ t'\colon\ C' \Rightarrow C''}{2\ c\vdash^*\ t\star t'\colon\ C \Rightarrow C''} \tag{8.42}$$

Fact 8.6.24 If 2 $c\vdash^*$ t: $C \Rightarrow C'$, also 1 $c\vdash^*$ t: $C \Rightarrow C'$.
If 2 $c\vdash$ t: $C \Rightarrow C'$, also 1 $c\vdash^*$ $\mathsf{P}(t)$: $\mathsf{P}(C) \Rightarrow \mathsf{P}(C')$. □

Proof: Induction in the derivation tree: the only interesting case is where (8.38) has been applied. We must show that

$$1\ c\vdash^*\ \mathsf{P}(\mathcal{I}_s(\mathrm{id}_{Ca_{H_1}} \& t \& \mathrm{id}_{Ca_{H_2}}))$$

But this is a consequence of fact 8.6.21. □
By combining lemma 8.6.23 and fact 8.6.24 we get

Fact 8.6.25 If 2 $c\vdash^*$ t: $C \Rightarrow C'$, there exists t_1, t_2, C'' such that

$$1\ c\vdash^*_u\ t_1\colon\ C \Rightarrow C'', 1\ c\vdash^*_f\ t_2\colon\ C'' \Rightarrow C', t = t_1\star t_2.$$

If C' is in normal form, $C' = C''$. □

8.6.8 Level 2 semantics

Definition 8.6.26 Given basic configuration B (not failure). Suppose 2 $c\vdash^*$ t: $B \Rightarrow C$ with C in normal form. Then $[\![\mathrm{B}]\!]_2 = C$.
If no such t and C exists, $[\![\mathrm{B}]\!]_2 = \perp$. □

By fact 8.6.25, this is well-defined.

We say that t is a fair level 2c single step if t takes the form $t = \mathcal{I}_s(t_1\& \ldots \& t_n)$, $n \geq 1$, each $t_i \in \mathcal{R}_1$. We say that t is a fair level 2c step if t takes the form $t = t_1 + \ldots + t_k$, at least one t_i being a fair level 2c single step and the rest of form id_B with B empty. We say that t is a fair level 2c unfolding if t takes the form $t = \mathsf{P}(t_1)\star \ldots \star\mathsf{P}(t_k)$, each t_i being a fair level 2c step. We say that B loops at level 2c by a fair strategy if for all $i > 0$ there exists C_i not in normal form and fair level 2c step t_i from C_{i-1} to C_i (here $C_0 = B$).

We say that t is a $\mathcal{LR}$ level 2c single step if t takes the form $t = \mathcal{I}_s(t_1 \& \mathrm{id}_B)$, with $t_1 \in \mathcal{R}_1$. We say that t is a $\mathcal{LR}$ level 2c step if t takes the form $t = t_1 + \ldots + t_k$, at least one t_i being a $\mathcal{LR}$ level 2 single step and the rest of form id_B with B empty. We say that t is a $\mathcal{LR}$ level 2c unfolding if t takes the form $t = \mathsf{P}(t_1)\star\ldots\star\mathsf{P}(t_k)$, each t_i being a $\mathcal{LR}$ level 2c step. We say that B loops at level 2c by the $\mathcal{LR}$ strategy if for all $i > 0$ there exists C_i not in normal form and $\mathcal{LR}$ level 2c step t_i from C_{i-1} to C_i (here $C_0 = B$).

Definition 8.6.27 Given basic configuration B (not failure). Suppose $2 \text{ c}\vdash^* t\colon B \Rightarrow C$ with C in normal form, where t is a $\mathcal{LR}$ level 2c unfolding. Then $[\![\mathrm{B}]\!]_2^L = C$.

If no such t and C exists, $[\![\mathrm{B}]\!]_2^L = \perp$. □

Clearly, $[\![\mathrm{B}]\!]_2^L = \perp$ iff B loops at level 2c by the $\mathcal{LR}$ strategy.

8.6.9 Total correctness

Theorem 8.6.28 Assume all U-mirrors occurring in rules in $\mathcal{R}_1$ satisfy $\mathcal{F}(1)$. Then for all B, $[\![\mathrm{B}]\!]_2 = [\![\mathrm{B}]\!]_1$. □

Proof: First suppose $[\![\mathrm{B}]\!]_2 = C \neq \perp$. By fact 8.6.25, also $[\![\mathrm{B}]\!]_1 = C$.

Now suppose $[\![\mathrm{B}]\!]_2 = \perp$. Then for all $n \geq 1$ there will exist fair level 2c step t_n and C_n not in normal form such that $2 \text{ c}\vdash^* \mathsf{P}(t_n)\colon C_{n-1} \Rightarrow C_n$ ($C_0 = B$). Let $t'_n = \mathsf{P}(t_1)\star\ldots\star\mathsf{P}(t_n)$. $2 \text{ c}\vdash^* t'_n\colon B \Rightarrow C_n$, and by fact 8.6.25 there exists t''_n, t'''_n and C'_n such that $1 \text{ c}\vdash^*_u t''_n\colon B \Rightarrow C'_n$, $1 \text{ c}\vdash^*_f t'''_n\colon C'_n \Rightarrow C_n$ and $t'_n = t''_n\star t'''_n$.

As C_n contains a non-empty basic configuration, this shows that t'_n for all n contains at least one mirror from B to a non-empty basic configuration. Then it will be possible (by Königs lemma) for all n to find $m_n \in t_n$ such that $m'_n = m_1\star\ldots\star m_n$ is a mirror in t'_n from B to a non-empty basic configuration. Also there will exist mirrors $m''_n \in t''_n$ and $m'''_n \in t'''_n$ such that $m'_n = m''_n\star m'''_n$. It is easily seen that $1 \vdash^*_u m''_n\colon B \Rightarrow B'_n$, with B'_n not failure nor empty.

Due to the assumption of the theorem, each m_i will satisfy $\mathcal{F}(1)$. Then, by lemma 8.3.32, each m'_n will satisfy $\mathcal{F}(n)$. But then also m''_n will satisfy $\mathcal{F}(n)$. By lemma 8.4.8, this shows that B loops at level 1 by a fair strategy, and by fact 8.6.18 $[\![\mathrm{B}]\!]_1 = \perp$. □

Theorem 8.6.29 Assume all U-mirrors occurring in rules in $\mathcal{R}_1$ satisfy $\mathcal{L}(1)$. Then for all B, $[\![\mathrm{B}]\!]_2^L \geq [\![\mathrm{B}]\!]_1^L$. □

Proof: First suppose $[\![\mathrm{B}]\!]_2^L = C \neq\perp$. Then also $[\![\mathrm{B}]\!]_2 = C$, so by fact 8.6.25 $[\![\mathrm{B}]\!]_1 = C$. Now either $[\![\mathrm{B}]\!]_1^L = C$ or $[\![\mathrm{B}]\!]_1^L =\perp$.

Now suppose $[\![\mathrm{B}]\!]_2^L =\perp$. Then for all $n \geq 1$ there will exist $\mathcal{LR}$ level 2c step t_n and C_n not in normal form such that $2\ \mathrm{c}\!\vdash^* \mathsf{P}(t_n)\colon C_{n-1} \Rightarrow C_n$ ($C_0 = B$). Let $t'_n = \mathsf{P}(t_1)\star\ldots\star\mathsf{P}(t_n)$. $2\ \mathrm{c}\!\vdash^* t'_n\colon B \Rightarrow C_n$, and by fact 8.6.25 there exists t''_n, t'''_n and C'_n such that $1\ \mathrm{c}\!\vdash^*_u t''_n\colon B \Rightarrow C'_n$, $1\ \mathrm{c}\!\vdash^*_f t'''_n\colon C'_n \Rightarrow C_n$ and $t'_n = t''_n\star t'''_n$.

As C_n contains a non-empty basic configuration, this shows that t'_n for all n contains at least one mirror from B to a non-empty basic configuration. Then it will be possible for all n to find $m_n \in t_n$ such that $m'_n = m_1\star\ldots\star m_n$ is a mirror in t'_n from B to a non-empty basic configuration. Also there will exist mirrors $m''_n \in t''_n$ and $m'''_n \in t'''_n$ such that $m'_n = m''_n\star m'''_n$. It is easily seen that $1 \vdash^*_u m''_n\colon B \Rightarrow B'_n$, with B'_n not failure.

Due to the assumption of the theorem, each m_i satisfies $\mathcal{L}(1)$. By lemma 8.3.37, there exists an increasing sequence

$$m'_1 \prec m'_2 \prec m'_3 \prec \ldots$$

Now two possibilities:

- $E(m'_n)$ is not bounded. Then neither $E(m''_n)$ is bounded, so by lemma 8.4.11 B loops at level 1 by a $\mathcal{LR}$ strategy.
- $E(m'_n)$ is bounded. Then $L(m'_n)$ is unbounded, so also $L(m''_n)$ is unbounded. Then again lemma 8.4.11 tells us that B loops at level 1 by a $\mathcal{LR}$ strategy.

In both cases, fact 8.6.20 tells us that $[\![\mathrm{B}]\!]_1^L =\perp$. □

This completes the technical development of the theory sketched in section 8.1.

Chapter 9

Concluding remarks

My hope is that the work reported in this thesis has been a contribution to the development of a more unified theory for program optimization, and that it has provided a better understanding of a wide range of phenomena – at least it has done so to me, of course it it not for me to judge whether it has been helpful for others too. In two respects, however, the treatment seems rather unsatisfactory:

- Only a *descriptive* point of view has been adopted; no tools for *analysis* have been presented. For instance such tools may be helpful in order to instantiate the USM properly (cf. chapter 5) or in order to assign weights in a suitable way (cf. chapter 8).
- It certainly seems waste of efforts (and space!) to have *two* models, one for a functional language (chapter 4) and one for a logic language (chapter 8). One should aim at a more unified treatment, i.e. to set up a *core model* which then can be instantiated appropriately.

These issues will be briefly addressed in the subsequent sections.

9.1 An analysis aiding the PEM

Concerning the development of analysis tools, we shall restrict ourselves to considering the following rather restricted problem:

> Given a source program p to be run by the PEM (cf. section 5.2). For each of the functions f_i occurring in p, suggest[1]

[1] As *all* choices are "safe", we shall prefer to use the term "heuristic" instead of "analysis".

which of the arguments to f_i should be static and which should be dynamic[2].

It is rather obvious that it will only pay off to make a certain subset of the arguments to f_i static provided the following two (rather loose) criteria are met:

1. these arguments assume the same value in several invocations of f_i;
2. it is possible to do "much" computation (i.e. unfoldings) without knowing the values of the remaining arguments.

There is a certain conflict between those two aspects, as 1 will become more likely if *few* arguments are static while 2 will become more likely if *many* arguments are static – what is "the best" choice is clearly undecidable in general.

Concerning 1, it seems rather hard to come up with heuristics giving a reasonable approximation in the general case. The most sensible thing to do is probably to "keep track of history" during run-time, similar to the well-known hack in the compiler industry: initially generate cheap but rather inefficient code; then observe in which part of the code most time is spent and recompile this piece of code using a more expensive method.

We shall now elaborate on 2, where a relatively useful heuristic seems feasible. The method will be sketched below, using the example from section 5.2.1 where two functions were present: *Int* taking arguments *e*, *p* and *d*; and *Branch* taking arguments *b*, *e2*, *e3*, *p* and *d*. The following reasoning can be performed:

1. As *e* is a needed argument (in the sense of section 4.3) of *Int* and *b* is a needed argument of *Branch*, these clearly have to be declared static.
2. In order for any interesting computation to be done by *Branch*, either *e2* or *e3* has to be static – otherwise only one unfolding is possible.

[2]Being a one-stage method, all arguments are present so in principle any choice is possible. On the other hand, in "classical" two-stage partial evaluation some parameters are simply not available "at PE-time" and therefore in principle have to be declared dynamic – nevertheless, using the fine art of *binding-time engineering* it might still be possible to declare such arguments static. . . .

3. As we cannot hope to predict the outcome of tests, the previous point suggests that *e2 as well as e3* have to be made static.

4. With the static arguments introduced so far, two rules may inhibit further unfolding: the rule for If and the rule for Rec. In order to enable further unfolding after the rule for Rec has been used, we have to declare p as a static argument to *Int* (and hence also to *Branch*). By the way, since the value of p does not change at all this will not diminish the chance of reusing computation, cf. criterion 1.

5. At the current stage, all arguments except d have been declared static. Still, after an application of the rule for If no further unfolding may take place, but in order to repair on this we have to declare *all* arguments static – and then it is doubtful whether computation can be reused at all.

We have thus argued that the choice of static arguments actually made in section 5.2.1 was not completely arbitrary...– in fact, the above line of reasoning should not be impossible to formalize and implement. That would certainly be an interesting area for further research.

Related approaches/remarks from the literature include

- a noteworthy part of the discussion following a talk by A.P. Ershov, reported in [Ers78, p. 420]:[3]

  > *Karel Culik*: In your program that computes X to the power n you assumed an input value for n equal to 5. Why didn't you assume that X was given an input value and n remained unspecified as 5 to the power n instead of X to the power 5?
  >
  > *Ershov*: I would like to stress that the reasons for arbitrary suspension are not formalized decisions like the one you would wish to be made on a formal basis. In this particular case one can see that X isn't an essential part of the computation, on the other hand n is processed bit by bit and so the program uses n "more intensively". But you would agree with me that it is not a mathematical consideration.

[3]The point is of course that it is possible to reduce X^5 to $X \cdot X \cdot X \cdot X \cdot X$, while 5^X cannot be reduced in a similar way.

- In [Lau91, Sect. 6], considering a first order functional language with parametric polymorphism, the observation is made that one does not gain anything by declaring arguments of polymorphic type static, since computation does not depend on such arguments.

- In the system described in [RW91], to each specialized function is associated information indicating how much of the static part which was actually *used* to produce the function – this enables specializations to be reused even though the (non-used) static arguments differ.

9.2 Integrating the functional and logical model

The functional and logic paradigms are not too far apart, as witnessed by several successful attempts at bridging the gap – e.g. [Red85] and [Han92].

So is there any excuse for my not coming up with a *core model*, capturing the abstract properties needed to formulate correctness/speedup theorems etc? Well, since "flow" of control and data differ in a substantial way some rather severe difficulties arise, cf. the discussion initially in chapter 8:

- ultimate sharing has no real counterpart in the logic world, due to the fact that a logic program can return *many* answers and a functional program only *one*;

- in the functional model an optimal evaluation strategy (cf. theorem 4.4.14) can be expressed syntactically without putting too tight restrictions on the form of the program. This is not the case in the logic world, where data flows around in a less controlled manner.

On the other hand, e.g. the speedup theorems are on a very abstract level; their essence is applicable to general theorem proving (therefore it seems highly plausible that they have appeared in other contexts).

The most promising path to bring the models closer together would probably be to transfer the idea of U-mirrors (cf. section 8.1.1) to the functional world, thus making it possible to express folding explicitly.

9.3 Miscellaneous

A major shortcoming of the work presented in this thesis is probably that it does definitely not pay too much attention to whether the insights gained are helpful to understand "large-scale" systems for program optimization. And even though I e.g. have played a bit with the Flagship system [DHK^{+}89], still (as admitted at the PLILP'92 conference!) I do not have any clear intuition about to which extent e.g. the unfold/fold methodology is useful in practical applications. For instance, the issue of how severe a restriction it is to only allow foldings which are abbreviations has been "swept in under the carpet" (like apparently most researchers in the field do!)

Bibliography

[Aho90] Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. A*, chapter 5. Elsevier, 1990.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Amt91] Torben Amtoft. Properties of unfolding-based meta-level systems. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9)*, 1991.

[Amt92a] Torben Amtoft. Unfold/fold transformations preserving termination properties. In M. Bruynooghe and M. Wirsing, editors, *4th International Symposium on Programming Language Implementation and Logic Programming (PLILP 92), Leuven, Belgium*, pages 187–201. Springer Verlag, LNCS no 631, August 1992.

[Amt92b] Torben Amtoft. Unfold/fold transformations preserving termination properties. Technical Report PB-410, DAIMI, University of Aarhus, Denmark, 1992. 58 pages.

[ANTJ89] Torben Amtoft, Thomas Nikolajsen, Jesper Larsson Träff, and Neil D. Jones. Experiments with implementations of two theoretical constructions. In *Logic at Botik, USSR (Springer LNCS no 363)*, pages 119–133, July 1989.

[AT89] Torben Amtoft and Jesper Larsson Träff. Memoization and its use in lazy and incremental program generation. Master's thesis, DIKU, University of Copenhagen, Denmark, August 1989. No 89-8-1.

[AT92] Torben Amtoft and Jesper Larsson Träff. Partial memoization for obtaining linear time behavior of a 2DPDA. *Theoretical Computer Science*, 98(2):347–356, May 1992.

[Bar84] H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, 1984.

[BCD90] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.

[BCE92] Annalisa Bossi, Nicoletta Cocco, and Sandro Etalle. On safe folding. In M. Bruynooghe and M. Wirsing, editors, *4th International Symposium on Programming Language Implementation and Logic Programming (PLILP 92), Leuven, Belgium*, pages 172–186. Springer Verlag, LNCS no 631, August 1992.

[BD77] R.M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.

[BDSK89] Maurice Bruynooghe, Danny De Schreye, and Bruno Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.

[Bel91] Francoise Bellegarde. Program transformation and rewriting. In *4th International Conference on Rewriting Techniques and Applications*, pages 226–239. Lecture Notes in Computer Science 488, 1991.

[Bir80] Richard S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[BKKS87] H.P. Barendregt, J.R. Kennaway, J.W. Klop, and M.R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75:191–231, 1987.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *ACM Symposium on Principles of Programming Languages*, pages 119–129, 1992.

[BvEG$^+$87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *PARLE, Eindhoven, The Netherlands. LNCS 259*, pages 141–158, 1987.

[CD89] Charles Consel and Olivier Danvy. Partial evalution of pattern matching in strings. *Information Processing Letters*, 30:79–86, January 1989.

[Cho90] Christian Choffrut. An optimal algorithm for building the Boyer-Moore automaton. *EATCS Bulletin no. 40*, pages 217–225, 1990.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[Coo71] Stephen A. Cook. Linear time simulation of deterministic two-way pushdown automata. In *Information Processing 71. Proceedings of IFIP Congress 1971*, pages 75–80. North-Holland, 1971.

[DHK$^+$89] John Darlington, Peter Harrison, Hessam Khoshnevisan, Lee McLoughlin, Nigel Perry, Helen Pull, Mike Reeve, Keith Sephton, Lyndon While, and Sue Wright. A functional programming environment supporting execution, partial execution and transformation. In *PARLE '89 (LNCS 365)*, pages 286–305, 1989.

[DP88] John Darlington and Helen Pull. A program development methodology based on a unified approach to execution and transformation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 117–131. North-Holland, 1988.

[DSMSB90] Danny De Schreye, Bern Martens, Gunther Sablon, and Maurice Bruynooghe. Compiling bottom-up and mixed derivations into top-down executable logic programs. In *Second Workshop on Meta-Programming in Logic, April 4-6, 1990, Leuven, Belgium. Ed.: M. Bruynooghe*, pages 37–56, 1990.

[Dyb85] Hans Dybkjær. Parsers and partial evaluation: An experiment. Student Project 85-7-15, DIKU, University of Copenhagen, Denmark, July 1985. 128 pages.

[ENRR87] H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors. *Graph-Grammars and Their Application to Computer Science*. Lecture Notes in Computer Science 291, 1987.

[Ers78] A.P. Ershov. On the essence of compilation. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[FS91] Gudmund S. Frandsen and Carl Sturtivant. What is an efficient implementation of the lambda-calculus? In John Hughes, editor, *International Conference on Functional Programming Languages and Computer Architecture*, pages 289–312. Springer Verlag, LNCS no 523, August 1991.

[GAL92] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *ACM Symposium on Principles of Programming Languages*, pages 15–26, 1992.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[Gre87] Steve Gregory. *Parallel Logic Programming in PARLOG - the language and its implementation*. Addison-Wesley, 1987.

[Gru] Klaus Grue. Call-by-mix: A reduction strategy for pure lambda-calculus. Circulated at DIKU, University of Copenhagen, Denmark.

[Gru87] Klaus Grue. An efficient formal theory. Technical Report 87/14, DIKU, University of Copenhagen, Denmark, 1987.

[GS91] P. A. Gardner and J. C. Shepherdson. Unfold/fold transformations of logic programs. In J.L. Lassez and G. Plotkin, editors, *Computational Proofs: Essays in honour of Alan Robinson.* 1991.

[Han92] Michael Hanus. Improving control of logic programs by using functional logic languages. In M. Bruynooghe and M. Wirsing, editors, *4th International Symposium on Programming Language Implementation and Logic Programming (PLILP 92), Leuven, Belgium*, pages 1–23. Springer Verlag, LNCS no 631, August 1992.

[Har87] David Harel. *Algorithmics - The Spirit of Computing.* Addison-Wesley, 1987.

[Hen87] Martin C. Henson. *Elements of Functional Languages.* Blackwell Scientific Publications, 1987.

[HG91] Carsten Kehler Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9)*, 1991.

[Hof92] Berthold Hoffmann. Term rewriting with sharing and memoization. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Volterra, Italy*, pages 128–142. Springer Verlag, LNCS no 632, September 1992.

[Hol91] Carsten Kehler Holst. Finiteness analysis. In John Hughes, editor, *International Conference on Functional Programming Languages and Computer Architecture*, pages 473–495. Springer Verlag, LNCS no 523, August 1991.

[Hon91] Zhu Hong. How powerful are folding/unfolding transformations. Technical Report CSTR-91-2, Department of Computer Science, Brunel University, January 1991.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Hug82] John Hughes. Super combinators - a new implementation method for applicative languages. In *ACM Symposium on Lisp and Functional Programming, Pittsburgh*, pages 1–10, 1982.

[Hug85] John Hughes. Lazy memo-functions. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 129–146. Springer Verlag, LNCS 201, 1985.

[Jon77] Neil D. Jones. A note on linear time simulation of deterministic two-way pushdown automata. *Information Processing Letters*, 6(4):110–112, August 1977.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Jør90] Jesper Jørgensen. Generating a pattern matching compiler by partial evaluation. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 177–195. Springer-Verlag, August 1990.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[Kah92] Stefan Kahrs. Unlimp, uniqueness as a leitmotiv for implementation. In M. Bruynooghe and M. Wirsing, editors, *4th International Symposium on Programming Language Implementation and Logic Programming (PLILP 92), Leuven, Belgium*, pages 115–129. Springer Verlag, LNCS no 631, August 1992.

[Kho90] Hessam Khoshnevisan. Efficient memo-table management strategies. *Acta Informatica*, 28:43–81, 1990.

[KK90] Tadashi Kawamura and Tadashi Kanamori. Preservation of stronger equivalence in unfold/fold logic program transformation. *Theoretical Computer Science*, 75:139–156, 1990.

[KMP77] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[Knu81] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, second edition, 1981.

[Kot80] Laurent Kott. A system for proving equivalences of recursive programs. In *Proceedings of 5th conference on Automated Deduction, Springer LNCS 87*, pages 63–69, 1980.

[Kot82] Laurent Kott. Unfold/fold program transformations. Technical Report 155, INRIA, Domaine de Voluceau Rocquencourt BP105 78153 Le Chesnay Cedex, France, 1982.

[Kot85] Laurent Kott. Unfold/fold program transformations. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 12. Cambridge University Press, 1985.

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *ACM Symposium on Principles of Programming Languages*, pages 16–30, 1990.

[Lau91] John Launchbury. Strictness and binding-time analyses: Two for the price of one. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada*, June 1991.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Mar91] Luc Maranget. Optimal derivations in weak lambda-calculi and in orthogonal terms rewriting systems. In *ACM Symposium on Principles of Programming Languages*, pages 255–269, 1991.

[Mic68] Donald Michie. 'Memo' functions and machine learning. *Nature*, 218:19–22, April 1968.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Nie89] Flemming Nielson. The typed lambda-calculus with first-class processes. In *PARLE '89 (LNCS 366)*, pages 357–373, 1989.

[Nie92] Flemming Nielson, editor. *Design, Analysis and Reasoning about Tools: Abstracts from the Second Workshop.* DAIMI PB-417, September 1992.

[NN90] Hanne Riis Nielson and Flemming Nielson. Eureka definitions for free! or disagreement points for fold/unfold transformations. In Neil D. Jones, editor, *ESOP 90, Copenhagen, Denmark. LNCS 432*, pages 291–305, May 1990.

[Pal89] Catuscia Palamidessi. Algebraic properties of idempotent substitutions. Technical Report TR-33/89, University of Pisa, 1989.

[PB82] Alberto Pettorossi and R.M. Burstall. Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique. *Acta Informatica*, 18:181–206, 1982.

[Pet84] Alberto Pettorossi. *Methodologies for Transformations and Memoing in Applicative Languages.* PhD thesis, University of Edinburgh, Department of Computer Science, October 1984.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, University of Aarhus, Denmark, September 1981.

[PP90] Maurizio Proietti and Alberto Pettorossi. Synthesis of eureka predicates for developing logic programs. In *Lecture Notes in Computer Science 432 (ESOP 90)*, pages 306–325, May 1990.

[PP91a] Maurizio Proietti and Alberto Pettorossi. Semantics preserving transformation rules for Prolog. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9)*, 1991.

[PP91b] Maurizio Proietti and Alberto Pettorossi. Unfolding - Definition - Folding, in this order, for avoiding unnecessary variables in logic programs. In *Proceedings of PLILP 91, Passau, Germany (LNCS 528)*, pages 347–358, August 1991.

[PP92] Maurizio Proietti and Alberto Pettorossi. Best-first strategies for incremental transformations of logic programs. In *Proceedings of LOPSTR 92, Manchester, 2-3 July 1992*, 1992.

[PS88] H. Partsch and F. A. Stomp. A formal derivation of Boyer and Moore's pattern matching algorithm. Technical Report 88-12, Department of Informatics, University of Nijmegen, September 1988.

[QG91] Christian Queinnec and Jean-Marie Geffroy. Symbolic pattern matching with intelligent backtrack. Can be achieved by email to (queinnec,geffroy)@poly.polytechnique.fr, 1991.

[Rao84] Jean Claude Raoult. On graph rewritings. *Theoretical Computer Science*, 32:1–24, 1984.

[Red85] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *IEEE Logic Programming Symposium, Boston*, pages 138–151, 1985.

[Red89] Uday S. Reddy. Rewriting techniques for program synthesis. In *Proceedings of 3rd International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science 355*, pages 388–403, 1989.

[RW91] Erik Ruf and Daniel Weise. Using types to avoid redundant specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9)*, 1991.

[Sah91] Dan Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Kungliga Tekniska Högskolan, Stockholm, SICS, Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden, March 1991.

[San90] David Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, Imperial College, London, September 1990.

[Sch80] William L. Scherlis. *Expression Procedures and Program Derivation.* PhD thesis, Stanford University, August 1980. Report STAN-CS-80-818.

[Sek91] Hirohisa Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86(1):107–139, 1991.

[Ses88] Peter Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. North-Holland, 1988.

[Smi91] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9)*, 1991.

[Søn89] Harald Søndergaard. Semantics-based analysis and transformation of logic programs. Technical Report 89/22, DIKU, University of Copenhagen, Denmark, 1989.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Sta80] John Staples. Optimal evaluations of graph-like expressions. *Theoretical Computer Science*, 10:297–316, 1980.

[Sun90] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, August 1990.

[Tak91] Akihiko Takano. Generalized partial computation for a lazy functional language. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9)*, 1991.

[TS84] Hisao Tamaki and Taisuke Sato. Unfold/fold transformation of logic programs. In *Proceedings of 2nd International Logic Programming Conference, Uppsala*, pages 127–138, 1984.

[Tur36] Alan Turing. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1936.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9:31–49, 1979.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Wad71] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Oxford University, 1971.

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[Wat80] Osamu Watanabe. Another application of recursion introduction. *Information Processing Letters*, 10(3):116–119, 1980.

[Yos93] Nobuko Yoshida. Optimal reduction in weak lambda calculus with shared environments. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 243–252. ACM Press, 1993.

Index