# Program Development by Proof Transformation

**Luca Chiarabini**

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
Ludwig–Maximilians–Universität
München

vorgelegt von
Luca Chiarabini


Luca Chiarabini

Program Development by Proof Transformation

Dissertation an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

1. Berichterstatter: Prof. Dr. Volker Heun
2. Berichterstatter: Prof. Dr. Helmut Schwichtenberg
3. Prüfer: Prof. Dr. Rolf Hennicker
4. Prüfer: Prof. Dr. Hans Jürgen Ohlbach

Ersatzprüfer: Prof. Dr. Martin Wirsing
Externer Gutachter: Prof. Dr. Stefano Berardi (Università di Torino)

With all its sinful doings, I must say,
That Italy's a pleasant place for me,
Who love to see the Sun shine every day,
And vines (not nail'd to walls) from tree to tree
Festoon'd, much like the back scene of a play
Or melodrame, which people flock to see
When the first act is ended by a dance
In vineyards copied from the south of France.

I also like to dine on becaficas,
To see the Sun set, sure he'll rise to-morrow,
Not through a misty morning twinkling weak as
A drunken man's dead eye in maudlin sorrow,
But with all Heaven t'himself; that day will break as
Beauteous as cloudless, nor be fore'd to borrow
That sort of farthing candlelight which glimmers
When reeking London's smoky cauldron simmers.

I love the language, that soft bastard Latin,
Which melts like kisses from a female mouth,
And sounds as if it should be writ on satin,
With syllables which breathe of the sweet South,
And gentle liquids gliding all so pat in,
That not a single accent seems uncouth,
Like our harsh northern whistling, grunting guttural,
Which we're oblig's to hiss, and spit, and sputter all.

I like the womens too (forgive my folly),
From the rich peasant-cheek of ruddy bronze,
And large black eyes that flash on you a volley
Of rays that say thousand things at once,
To the high dama's brow, more melancholy,
But clear, and with a wild and liquid glance,
Heart on her lips, and soul within her eyes,
Soft as her clime, and sunny as her skies.

Con tutti i suoi peccati, devo dire
Che l'Italia mi piace, che mi piace
Vedere il sole splendere ogni giorno,
E le viti non piantate su un muro,
Ma abbarbicate ai tralicci, fondi
D'opera dove la gente accorre
Quando una danza chiude il primo atto,
Tra vigne rosseggianti come in Francia.

Mi piace poi mangiare beccafichi,
Guardare il sole che tramonta, certo
Che domani risorge e non opaco
Come un occhio ubriaco tra le nubi,
Ma in pieno cielo rinascerà il giorno,
Lucente e senza nuvole, e non gonfio
Di quel torvo lucore di candela
Del fetido bollore londinese.

La lingua, poi, quel latino bastardo
Morbido come il bacio di una donna,
Che vibra come se scritto sul raso,
Sillabe respiranti il mezzogiorno,
Le liquide che scorrono gentili,
Dove nessun accento suona rozzo
Come le gutturali nordiche, grugniti
O fischi che sputiamo, scoppiettanti.

Infine (perdonate) amo le donne,
Le ricche guance contadine bronzee,
E gli occhi neri, e irradianti, e grandi,
Che ti dicono tutto in un istante,
Le dame, la fronte malinconica,
Ma chiara e dallo sguardo selvatico,
Cuore su labbra, sugli occhi l'anima,
Solare e dolce come il cielo e il clima.

Da *Beppo: A Venetian Story [Beppo: Una Storia Veneziana]* di Lord Byron

## Abstract

In the last 20 years the formal approach to the development of software turned out to be a crucial technique for the generation of correct programs.

This idea has its theoretical base into the several semi-automatic methods to transform a formal specification that describe the behavior of a program into an effective executable piece of code.

One of this is the so-called "program extraction from proof". The idea is that from an constructive proof of a formula "for each x there exists y such that P(x,y)" we can automatically extract a program "t" such that the property P(x,t(x)) hold. In our days such proofs are normally written by ad-hoc tools (some of them are: COQ, ISABLLE, MINLOG, PX, AGDA, etc...) called "proof assistants".

Even if today this technique is pretty well established, the "manipulation" of proofs in order to develop performing programs did not received big attention. In this thesis we will develop several automatic and semi-automatic methods in order to extract efficient code from constructive proofs. Our field of application will be computational biology, a research field in which the development of efficient programs is crucial. So our main goal will be to show how the manipulation of formal proofs, essentially studied by proof theorist, has a big effect also in practical program generation.

In den letzten 20 Jahren stellte sich der Einsatz formaler Methoden in der Softwareentwicklung als eine äußerst wichtige Technik zur Generierung korrekter Programmen heraus.

Die theoretische Grundlage dieser Idee basiert auf mehreren semiautomatischen Methoden zur Umwandlung einer formalen Spezifizierung, die das Verhalten eines Programms beschreibt, zu einem ausführlichen Codeblock.

Eine dieser Methoden nennt sich "program extraction from proof". Die Idee ist, dass wir von einem konstruktiven Beweis einer Formel “für jedes x existiert ein y so dass P(x,y)” ein Programm “t” automatisch extrahieren kann, in welchem die Eigenschaft P(x,t(x)) erfüllt ist. Heutzutage werden solche Beweise von ad hoc Tools erzeugt (z.B.: COQ, ISABLLE, MINLOG, PX, AGDA, usw.), die “proof assistants” genannt werden.

Obwohl sich diese Technik heutzutage gut etabliert hat, hat die “Manipulation” von Beweisen, mit den Ziel effiziente Programme zu realisieren, keine große Beachtung gefunden. Innerhalb dieser Doktorarbeit werden wir verschiedene automatische und semiautomatische Methoden mit dem Ziel entwickeln, Code von konstruktiven

Beweisen zu extrahieren. Unser Anwendungsbereich wird die Bioinformatik

sein, ein Forschungsbereich für den die Entwicklung effizienter Programme entscheidend ist. Unser Ziel wird folglich sein zu zeigen, wie die Manipulation von formalen Beweisen - hauptsä chlich erforscht von Beweistheoretikern - eine große Auswirkung auf die praktische Programmgenerierung hat.

# Acknowledgements

I wish to thank my advisors, Prof. Dr. Helmut Schwichtenberg and Prof.Dr. Volker Heun for the help and the guidance which they have given to me during all the period of my doctoral studies. Thanks also to my external advisor Stefano Berardi for helping me during the corrections of the thesis. I wish to thank my colleagues Diana Ratiu, Stefan Schimanski, Freiric Barral, Trifon Trifonv, Bogomil Kovachachev, Basil Karàdais and Simon W. Ginzinger for their academic and human support. I wish to thank Frau. Gerlinde Bach for the help in filling dozens of bureaucratic documents and Franziska Schneider for the nice philosophical chats.

I wish to thanks all the members of the "pataccas" or "pizzas" Munich group: Antonio Marraffa, Giuseppe Marraffa, Giovanni Alunni, Simone Brenner, Mauro Improta, Martina Dreißig, Marco Favorito, Manuela Bianchi, Rocco Marvaso and Agostino Santisi. I passed with them wonderful moments. I think that, without them, I would never ever had the power to live in Munich for so long time. They are and will remain my best friends. I wish also to thank all the friends in the Internationale Haus of Munich (one of the most exciting place I ever had the chance to live) in particular to Tonia Ludwig for her kindness. Thanks to the Genova's friends, among them David Burlando and Luana Noselli for all the support they gave to me in the last five years.

I wish to thank all my relatives: my grandmothers Franca and Iolanda, my cousins Francesca, Katy, Emanuele and Elisa, my uncles Eugenio, Paolo, Roberto and my aunts Mara, Rosa and Serenella. A special thought goes to my grandparents Anselmo and Angelo, that left us too early.

Finally, I wish to thank the most important persons in my life, the persons without whom I would not be here today and that always supported me: my mother Loredana Grassellini and my father Valter Chiarabini, to which this thesis is dedicated.

# Contents

# 1 Introduction

## 1.1 Automatic Program Development

The software life-cycle [26] (Figure 1.1) is the our-days model for the production of software in the industrial world. The basic idea is the following: given an input problem (most of the time specified in natural language -as English-) one write a program that is assumed to solve the problem. Afterwards the program is tested on several inputs and modified in case errors pop up. After this step, the program is put in practical use.

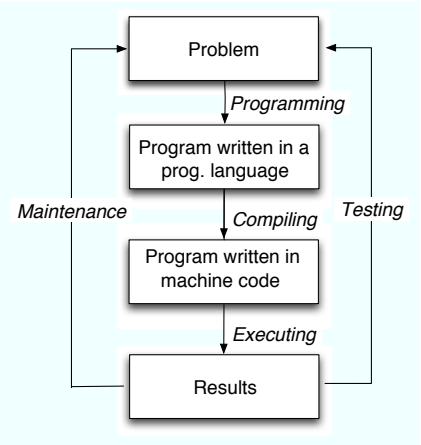



Figure 1.1: A software life-cycle model illustrating conventional software design

The main limit of this approach is that it can only confirm the presence of errors but not their absence. What we miss following this approach is the evidence of the correctness of the program. A better methodology for the production of correct software with respect to a given specification, rely on deriving a program from a problem in several controlled steps as illustrated Figure 1.2.

The step in Figure 1.2 can be in the following way resumed:

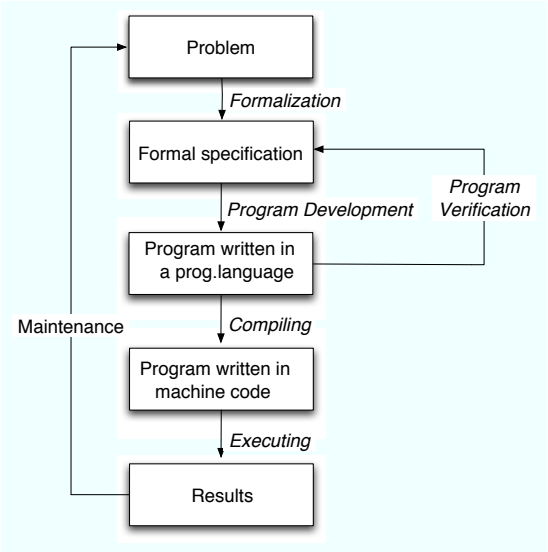



Figure 1.2: A software life-cycle model illustrating conventional software design

1. The problem of the customer is analyzed and a first *informal* specification is produced.
2. The formal specification is translated in a more formal language (equational for term rewriting, or Horn-clausal form for logic programming)
3. From the formal specification is derived a program that is provably correct, that is can be proven that the program meets the specification (program verification).
4. The derived program can be compiled and executed and the results can be used to test the program.

Essentially, there are two broad paradigms to fulfill step number 3: the "proofs-as-programs"[2] and "synthesis by transformations" [8].

- In the proof-as-program paradigm a specification is usually expressed by formulas that state the existence of an object with a given property. Thus a constructive proof of the given specification is produced and a program is extracted from the proofs. By the realizability method we can prove

that the program so produced respects the given specification (that is the proved formula). Research in this field focuses on the development of strong theorem provers and mechanisms for extracting algorithms from proofs.

- In synthesis by transformations the algorithms are derived from the specification by forward reasoning. The specification is seen as executable and is transformed in a real program by a set of rewriting rules. This paradigm is particularly well-suited for the synthesis of logic programs since a declarative formula can be viewed as executable program which "only" has to be transformed into some restricted syntax like Horn logic.

Our work concerned essentially the proof-as-program paradigm. According to this paradigm we have the following correspondences

$$\begin{aligned} \text{formula} &\equiv \text{data type} \\ \text{constructive proof of formula } A &\equiv \text{program of type } A \end{aligned}$$

The basic idea in order to develop correct programs by the proof-as-program methodology can be resumed in the followings steps:

- We assume that the programming problem is given in the form

$$\forall x \exists y A(x, y)$$

- One finds (manually, or computer-aided) a constructive formal proof of the formula $\forall x \exists y A(x, y)$.

- From the proof a program $p$ is extracted (fully automatically) that provable meets the specification, that is,

$$\forall x A(x, p(x))$$

  is provable

There exist a number of systems supporting program extraction from proofs (e.g. Agda[1], Coq[2], Minlog[3], NuPrl[4]).

From the end of the '80s a lot of research focused on the development of efficient algorithms by the proof-as-programs paradigm. This was stimulated by the fact that often the computational content of elegant and short proofs is

[1] http://unit.aist.go.jp/cvs/Agda/
[2] http://coq.inria.fr/
[3] http://www.minlog-system.de/
[4] http://www.cs.cornell.edu/Info/Projects/NuPRL/nuprl.html

particularly inefficient. Consider for example the following statement:

*For each natural number $n$ there exists a natural $y$ such that $y = 2^n$.*

This sentence is simply provable by induction on $n$. In the base case its enough to set $y = 1$, in fact $1 = y = 2^0$. Then if (by induction hypothesis) we know that $\overline{y} = 2^n$ for some fixed $n$, to prove the sentence for $n+1$ its enough to to set $y = \overline{y} + \overline{y}$. In fact

$$\begin{aligned} y &= \overline{y} + \overline{y} \\ &= 2^n + 2^n \\ &= 2^{n+1} \end{aligned}$$

By the proof-as-program paradigm the computational content of this proof is the *power of 2* function, $\mathsf{EXP}_2$, sketched in the following piece of code:

**Algorithm 1** Procedure $\mathsf{EXP}_2$

**Input:** $0 \leq n$
**Output:** $2^n$
**loop**
  **if** $n = 0$ **then**
    **return** 1
  **else**
    **return** $\mathsf{EXP}_2(n-1) + \mathsf{EXP}_2(n-1)$
  **end if**
**end loop**

Unfortunately the computational complexity of $\mathsf{EXP}_2$ is exponential in $n$. Historically the research concerning the problem of extracting efficient programs from proofs focused both in tuning the extracted code[12, 3, 7] (the optimization phase take place after the extraction) and in tuning the proof from which the code is extracted[30, 29, 1] (the optimization phase take place before the extraction) Our work regarded this second line of research.

## 1.2 Content of the Thesis

The originality of the present work regarded the development of a set of new proof-techniques to transform proofs in order to develop efficient programs. In particular we investigated and developed the following proof-transformations:

**Pruning** This technique has its theoretical bases in the proof theory work of Dag Prawitz [31] later on successfully developed in the pioneer work of

C.A. Goad [17]. Pruning regards the eliminations of redundant case distinctions in proofs. Consider for example the following simple statement:

*Given a natural $n$ there exists a natural $y$ such that $n \leq y$*

We can prove this statement as follow. We assume $n$. There are two cases: $n \leq 1$ or $n \not\leq 1$. Assume $\mathsf{ip} : n \leq 1$ then we set $y = 1$, and we have the thesis by $\mathsf{ip}$. Else (that is $n \not\leq 1$) we set $y = n$ and we conclude by the reflexivity of the less-or-equal relation between naturals numbers. The computational content of this proof is the following piece of code:

**Algorithm 2**

**Input:** $0 \leq n$
**Output:** $0 \leq y$ such that: $n \leq y$
  **if** $n \leq 1$ **then**
    **return** 1
  **else**
    **return** $n$
  **end if**

Of course in the above proof the case distinction over $n$ is useless (we could for example immediately conclude setting $y = n$). The pruning technique is useful in detecting and simplifying this kind of redundancies. The main idea on which pruning is based is the following: if the left/right branch of a case distinction proof over $A \vee B$ does not depend on the assumptions $A/B$, then the entire case distinction can be replaced by the left/right branch.

In the example above, the left branch of the case distinction refer to the assumption variable $u : n \leq 1$, but the right branch does not depend on the condition $n \not\leq 1$. So applying the pruning rule, we can replace the case distinction by its right branch, obtaining a new proof from which we can extract the identity function. We note as the simplified extracted program is not only more efficient (we don't perform a useless "if") but it changes also its computational behavior.

In the chapter 3 of this thesis we extensively revisit the pruning idea and we apply it in simplifying some instantiations of the proof of the bin packing problem. In chapter 4 we develop a proof of the bounded perfect matching problem and we simplify some instantiations of it with the pruning technique, showing on another not trivial example that pruning has to be considered an essential tool in order to extract efficient programs from instantiated proofs. Finally in chapter 5 we prose an extension of

pruning.

**Dynamic Programming** The question that motivated this line of research was the following: how it is possible to transform a proof into another proof, from which it is possible to extract a dynamic program? We refer to dynamic programming as a programming technique where we evaluate a sufficient amount of data in advance so that the at each iteration the program gets to reuse it instead of recomputing it each time it is needed. Though at programming level this technique is pretty well known, it is not so clear how to obtain the same result at proof level. In chapter 6 of the thesis we developed (taking as a case study the formalization of the similarity of DNA sequences problem) a general method in order to extract dynamic programs from proof. The proposed method unfortunately is not general enough to be applied automatically to a large set of proofs (the automatic transformation is not possible even at programming level). What we developed has to be considered more as a general scheme that should instantiated case by case.

In order to get an informal idea of the method (that will be formally presented in chapter 6), let consider the following example. Assume we want to prove, for each $0 \leq n$, the existence of a natural $y$ such that $y = \mathsf{Fib}(n)$ with $\mathsf{Fib}(n)$ $n$-th Fibonacci number, defined as usual:

$$\mathsf{Fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathsf{Fib}(n-1) + \mathsf{Fib}(n-2) & 2 \leq n \end{cases}$$

This statement can be proved by (general) induction over $n$ as follow: for $n = 0$ we set $y = 0$, for $n = 1$ we set $y = 1$ and for $2 \leq n$, we apply the induction hypothesis

$$\forall n(\forall k.k < n \rightarrow \exists y.y = \mathsf{Fib}(k))$$

on $n-1$ and $n-2$ obtaining $u_1 = \mathsf{Fib}(n-1)$ and $u_2 = Fib(n-2)$ and thus we have the thesis for $y = u_1 + u_2$. In Algorithm 3 is showed the computational content of this proof.

In Algorithm 3 the procedure fib has an exponential computational complexity in $n$. The idea we propose in this thesis to tune this kind of proof (in order to extract dynamic programs) consist in adding a set of new axioms to manage a list of intermediate computed results in order to avoid re-computation. For example for the specific case of the Fibonacci numbers, an idea would be to introduce a new predicate $\mathtt{MEM} \subseteq \mathbf{N} \times \mathbf{N} \times \mathbf{N}$, where $\mathtt{MEM}(i, f_{i-1}, f_i)$ (for $1 \leq i$) means that $f_{i-1}$ and $f_i$ are the $i-1$-th

**Algorithm 3** Procedure fib

**Input:** $0 \leq n$
**Output:** $\mathsf{Fib}(n)$

```
loop
  if n = 0 then
    return 0
  else if n = 1 then
    return 1
  else
    return fib(n − 1) + fib(n − 2)
  end if
end loop
```

and $i$-th Fibonacci number. The axioms required in this case would be necessary to state formally that the value we store in $f_{i-1}$ and $f_i$ are Fibonacci numbers. Then new thesis to prove require a little modification: we have to show that for each natural $n$ there exists a natural $y$ such that $y = \mathsf{Fib}(n)$ *and* that there exists two naturals $w$ and $z$ such that $\texttt{MEM}(n, w, z)$. Later on, in the proof of the new thesis, we can avoid to instantiate twice the induction hypothesis (source of the exponential behavior of fib) and we can refer to the induction hypothesis only once and to the partial results *stored* in `MEM`. The computational content of this proof is a linear time algorithm.

**Tail Recursion** For a program to be *tail recursive* is a desired property that guarantee a certain level of efficiency. In a tail recursive procedure the recursive call are done as last operation: this avoid, during the compilation or interpretation task, the storage/recover (during the call/return of the procedure) of a big amount of data (the procedure-contexts). One of the main tool to perform an automatic transformation of a program into a tail recursive one is the so called CPS [33] [13] (Continuation Passing Style) transformation.

In the chapters 7 and 8 of the present thesis we investigated the relation between constructive proofs and tail recursion. In particular, our study was motivated by the following question: how it is possible to transform (possibly automatically) a proof by induction into another proof in such a way the content of the transformed proof is tail recursive?

In the literature, one of the main references (that we will briefly review later) on this topics, is the Penny Anderson's Ph.D. thesis [1]. Though the approach described in the Anderson's thesis is extremely interesting,

this is not completely automatic but it require some user interaction. In the present thesis we develop a method fully automatic to obtain the same result, based on a particular simple idea.

Let consider for example the task to prove that for each natural $n$ there exist a natural $y$ such that $y = \mathsf{Fact}(n)$ with $\mathsf{Fact}(n)$ the factorial of $n$ defined as follow :

$$\mathsf{Fact}(n) = \begin{cases} 1 & n = 0 \\ n * \mathsf{Fact}(n-1) & 0 < n \end{cases}$$

We can prove this statement by induction on $n$. For $n = 0$ we set $y = 1$ and assuming $u = \mathsf{Fact}(n)$ then we can build the factorial of $n+1$ setting $y = n * u$. The content of this proof is the usual factorial function in Algorithm 4.

**Algorithm 4** Procedure fact

**Input:** $0 \leq n$
**Output:** $\mathsf{Fact}(n)$

```
loop
  if n = 0 then
    return 1
  else
    return n * fact(n − 1)
  end if
end loop
```

The procedure fact in Algorithm 4 is not tail recursive (in the **else** branch we have to store the context $(n * \_)$). An idea to tune fact is to shift the control of the execution to another recursive procedure that will be tail-called and use an accumulator parameter where the effective computation of the factorial numbers will take place. At logical level this is done by proving an intermediate lemma, where we state that, given two naturals $n$ and $m$ and the the factorial for $m$, $u = \mathsf{Fact}(m)$, we are able to supply a natural $y$ such that $y = \mathsf{Fact(n+m)}$. The proof of this intermediate lemma is the heart of the transformation and it will be carefully presented in chapter 8. Later on, we can instantiate the proof of this lemma on a generic $n$ and on 0 in order to obtain the proof of the factorial of $n$. In this example, we worked with the factorial of $n$ but it is possible to apply the method to a generic predicate $P(n)$.

## 1.3 Related Work

We can divide the literature in the field of the generation of efficient programs by the usage of a proof assistants into two big blocks: methods that transform a program after the extraction phase (I), and methods to transform a proof in order extract efficient code (II).

(I)

In [28], Nakoi Kobayashi propose a method to solve the "useless-variable elimination" problem. This is one of the problems that affect the code automatically extracted from a proof. The proposed algorithm to solve the problem is a surprisingly simple extension of the usual type-reconstruction algorithm. The proposed method has several attractive features. First, it is simple, so that the proof of the correctness is clear and the method can be easily extended to deal with a polymorphic language. Second, it is efficient: for a simply-typed $\lambda$-calculus, it runs in time almost linear in the size of an input expression.

In [3] Stefano Berardi presents a pruning method to simplify program extracted from proofs. The proposed method is based on the replacement of some sub-terms with dummy constants. Berardi proves that the proposed method preserves observational behavior of a simply typed $\lambda$-term if it does not modify the type nor the context (assignment of types to free variables) of the term. This result is used to define a map Fl : simply typed $\lambda$-terms $\rightarrow$ simply typed $\lambda$-terms removing redundant code in functional programs. In the paper are formally proved some properties of Fl interesting from a computational viewpoint.

In [12], Damiani and Giannini presents two type inference systems for detecting useless-code in higher-order typed functional programs. This work represents an extension of the previously analyzed work of Berardi on pruning. In the paper it is proposed a useless-code elimination algorithm which is based on a combined use of these type inference systems. The main application of the technique is the optimization of programs extracted from proofs in logical frameworks, but it can be used as well in the elimination of useless-code determined by program transformations.

(II)

In [17] Alan Goad introduce the use of the pruning for the development of efficient programs generated by formal proofs. The paper concerns: (1) the uses of this additional information in the automatic transformation of algorithms, and in particular, in the adaptation of algorithms to special situations, and (2) efficient methods for executing and transforming proofs. The proposed method is later on tested on the implementation of the bin packing problem.

In [1], Penny Anderson propose a solution to the problem of transforming

a proof in order to extract a tail recursive function. The method is based on the representation of derived logical rules in Elf, a logic programming language that gives an operational interpretation to the Edinburg Logical Framework. It results in declarative implementations with a general correctness property that is verified automatically by the Elf type checking algorithm.

In [30] Frank Pfenning presents an interesting proof transformation to extract efficient code from proofs (this work constitute the theoretical base of the Anderson's work [1]). In his paper Pfenning extends the paradigm employed in systems like NuPrl where a program is developed and verified through the proof of the specification in a constructive type theory. The method is illustrated on an extended example – a derivation of Warshall's algorithm for graph reachability. In the paper, the author, outline how the framework supports the definition, implementation, and use of abstract data types.

# 2 Logical Foundations

## 2.1 Modified Realizability for First Order Minimal Logic

### 2.1.1 Gödel's T

Types are built from base types $\mathbf{N}$ (Naturals) , $\mathbf{L}(\rho)$ (lists with elements of type $\rho$) and $\mathbf{B}$ (booleans) by function ($\to$) and pair ($\times$) formation. The *Terms* of Gödel's T [39] are simply typed $\lambda$-calculus terms with pairs, projections ($\pi_i$) and constants (constructors and recursive operators for the basic types)

$$\begin{array}{rrl} \textit{Types } \rho,\sigma & ::= & \mathbf{N} \mid \mathbf{B} \mid \mathbf{L}(\rho) \mid \rho \to \sigma \mid \rho \times \sigma \\ \textit{Const } c & ::= & 0^{\mathbf{N}} \mid \mathsf{Succ}^{\mathbf{N}\to\mathbf{N}} \mid \mathsf{tt}^{\mathbf{B}} \mid \mathsf{ff}^{\mathbf{B}} \mid (:)^{\mathbf{L}(\rho)} \mid ::^{\rho\to\mathbf{L}(\rho)\to\mathbf{L}(\rho)} \mid \mathcal{R}^{\sigma}_{\mathbf{N}} \mid \mathcal{R}^{\sigma}_{\mathbf{L}(\rho)} \mid \mathcal{R}^{\sigma}_{\mathbf{B}} \\ \textit{Terms } r,s,t & ::= & c \mid x^{\rho} \mid (\lambda x^{\rho} r^{\sigma})^{\rho\to\sigma} \mid (r^{\rho\to\sigma} s^{\rho})^{\sigma} \mid (\pi_0 t^{\rho\times\sigma})^{\rho} \mid (\pi_1 t^{\rho\times\sigma})^{\sigma} \mid (r^{\rho}, s^{\sigma})^{\rho\times\sigma} \end{array}$$

The expression (:) represents the empty list, and $(a_0 :: \ldots :: a_n :)$ a list with $n+1$ elements. We equip this calculus with the following usual conversion rules for the recursive operators, applications and projections:

$$\begin{array}{ll} \mathcal{R}^{\sigma}_{\mathbf{N}} : \sigma \to (\mathbf{N} \to \sigma \to \sigma) \to \mathbf{N} \to \sigma & \mathcal{R}^{\sigma}_{\mathbf{L}(\rho)} : \sigma \to (\rho \to \mathbf{L}(\rho) \to \sigma \to \sigma) \to \mathbf{L}(\rho) \to \sigma \\ (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)\, 0 \longmapsto b & (\mathcal{R}^{\sigma}_{\mathbf{L}(\rho)}\, b\, f)\, [] \longmapsto b \\ (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)\, (n+1) \longmapsto f\, n\, ((\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)\, n) & (\mathcal{R}^{\sigma}_{\mathbf{L}(\rho)}\, b\, f)\, (a :: l) \longmapsto f\, l\, ((\mathcal{R}^{\sigma}_{\mathbf{L}(\rho)}\, b\, f)\, l) \end{array}$$

$$\begin{array}{ll} \mathcal{R}^{\sigma}_{\mathbf{B}} : \sigma \to \sigma \to \mathbf{B} \to \sigma & \pi_0(r,s) \longrightarrow r \\ (\mathcal{R}^{\sigma}_{\mathbf{B}}\, r\, s)\mathsf{tt} \longmapsto r & \pi_1(r,s) \longrightarrow s \\ (\mathcal{R}^{\sigma}_{\mathbf{B}}\, r\, s)\mathsf{ff} \longmapsto s & (\lambda x.r)s \longrightarrow r[x := s] \end{array}$$

### 2.1.2 Heyting Arithmetic

We define Heyting Arithmetic $\mathrm{HA}^{\omega}$ for our language based on Gödel's T, which is finitely typed.

*Formulas*: Atomic formulas ($\mathrm{P}\vec{t}^{\vec{\rho}}$) (P a predicate symbol, $\vec{t}$, $\vec{\rho}$ lists of terms and types), $A \to B$, $\forall x^{\rho} A$, $\forall^{\mathsf{nc}} x^{\rho} A$, $\exists x^{\rho} A$, $\exists^{\mathsf{nc}} x^{\rho} A$, $A \wedge B$. Given a term $t$ of type $\mathbf{B}$ we define a special kind of atomic formula, $\mathtt{atom}(t)$ that means '$t = \mathsf{tt}$'. In particular we have the atomic formula $\bot := \mathtt{atom}(\mathsf{ff})$. We define *negation* $\neg A$ by $A \to \bot$. In writing formulas we assume that $\forall, \exists, \neg$ bind more strongly than $\wedge$, and that in turn $\wedge$ binds more strongly than $\to$.

*Derivations*: By the Curry-Howard correspondence it is convenient to write derivations as terms: we define $\lambda$-terms $M^A$ for natural deduction proofs in *minimal logic* of formulas $A$ together with the set $\mathrm{OA}(M)$ of *o*pen *a*ssumptions in $M$:

| | |
|---|---|
| $(ass)$ | $u^A$, $\mathrm{OA}(u)=\{u\}$ |
| $(\wedge^+)$ | $(\langle M^A, N^B\rangle^{A\wedge B})$, $\mathrm{OA}(\langle M,N\rangle)=\mathrm{OA}(M)\cup\mathrm{OA}(N)$ |
| $(\wedge_0^-)$ | $(M^{A\wedge B}0)^A$ , $\mathrm{OA}(M0)=\mathrm{OA}(M)$ |
| $(\wedge_1^-)$ | $(N^{A\wedge B}1)^B$ , $\mathrm{OA}(N1)=\mathrm{OA}(N)$ |
| $(\to^+)$ | $(\lambda u^A M^B)^{A\to B}$ , $\mathrm{OA}(\lambda uM)=\mathrm{OA}(M)\backslash\{u\}$ |
| $(\to^-)$ | $(M^{A\to B}N^A)^B$, $\mathrm{OA}(MN)=\mathrm{OA}(M)\cup\mathrm{OA}(N)$ |
| $(\forall^+)$ | $(\lambda x^\rho M^A)^{\forall x^\rho A}$, $\mathrm{OA}(\lambda xM)=\mathrm{OA}(M)$ |
| | provided $x^\rho\notin \mathrm{FV}(B)$, for any $u^B\in\mathrm{OA}(M)$ |
| $(\forall^-)$ | $(M^{\forall x^\rho A}t^\rho)^A$, $\mathrm{OA}(Mt)=\mathrm{OA}(M)$ |
| $(\forall^{\mathsf{nc}+})$ | $(\lambda^{\mathsf{nc}} x^\rho M^A)^{\forall^{\mathsf{nc}} x^\rho A}$, $\mathrm{OA}(\lambda^{\mathsf{nc}} xM)=\mathrm{OA}(M)$ |
| | provided $x^\rho\notin \mathrm{FV}(B)$, for any $u^B\in\mathrm{OA}(M)$, and $x\notin[\![M]\!]$ |
| $(\forall^{\mathsf{nc}-})$ | $(M^{\forall^{\mathsf{nc}} x^\rho A}t^\rho)^A$, $\mathrm{OA}(Mt)=\mathrm{OA}(M)$ |

To obtain *intuitionistic logic* we can use the additional *ex-falso-quodlibedt* rule:

$$\forall\vec{x}(\bot\to P(\vec{x})) \qquad (\mathsf{Efq})$$

with $P$ predicate symbol different from $\bot$. We will use two specials quantifiers $\forall^{\mathsf{nc}}/\exists^{\mathsf{nc}}$ to indicate that there should be **n**o **c**omputational **c**ontent [5][4]. The logical meaning of the universal quantifiers is unchanged. However, we have to observe a special *variable condition* for $\forall^{\mathsf{nc}+}$: the variable to be abstracted should not be a *computational variable* in the given proof, i.e. the extracted program of this proof should not depend on $x$.

We will write proofs in form of proof-terms, as above, or as metarules

$$\frac{A_1,\ldots,A_n}{C}\,\mathcal{R}$$

to read as 'from the assumptions $A_1,\ldots,A_n$, by the rules $\mathcal{R}$ we derive $C$. Here $\mathcal{R}$ can be an introduction rule ($\wedge^+$, $\to^+$, $\forall^+$,$\forall^{\mathsf{nc}+}$) or an elimination rule ($\wedge_0^-$, $\wedge_1^-$, $\to^-$, $\forall^-$, $\forall^{\mathsf{nc}-}$).

Usually we will omit type and formula indices in derivations if they are uniquely determined by the context or if they are not relevant. We use $\exists$ (with or without computational content) and $\vee$ in our logic, if we allow the following axioms as constant derivation terms:

$$\exists^+_{x^\rho,A} \quad : \quad \forall x^\rho(A\to\exists x^\rho A)$$

$$\begin{array}{rcl}
\exists^-_{x^\rho,A,B} & : & \exists x^\rho A \to \forall x^\rho(A \to B) \to B \text{ with } \ x^\rho \notin FV(B) \\
(\exists^{\mathsf{nc}})^+_{x^\rho,A} & : & \forall^{\mathsf{nc}} x^\rho(A \to \exists^{\mathsf{nc}} x^\rho A) \\
(\exists^{\mathsf{nc}})^-_{x^\rho,A,B} & : & \exists^{\mathsf{nc}} x^\rho A \to \forall^{\mathsf{nc}} x^\rho(A \to B) \to B \text{ with } \ x^\rho \notin FV(B)
\end{array}$$

The constant $\exists^-$ followed by $\exists^+$ *elide* themselves by the following ($\mathsf{Elid}$) rule:

$$\exists^-(\exists^+ t^\rho\, M^A)^{\exists x^\rho A} = \lambda y^{\forall x^\rho A \to B}.y\, t^\rho\, M^A$$

We can define $\vee$ from $\exists$ via:

$$A \vee B \triangleq \exists p^{\mathbf{B}}.(p \to A) \wedge ((p \to \bot) \to B)$$

Here (for short) we wrote $p$ for $\mathsf{atom}(p)$. The induction proof-terms associated with $\mathbf{N}$,$\mathbf{B}$ and $\mathbf{L}(\rho)$ are:

$$\begin{array}{rcl}
\mathrm{Ind}_{n,A(n)} & : & A(0) \to (\forall n.A(n) \to A(n+1)) \to \forall n^{\mathbf{N}}.A(n) \\
\mathrm{Ind}_{t,A(t)} & : & A(\mathsf{tt}) \to A(\mathsf{ff}) \to \forall t^{\mathbf{B}}.A(t) \\
\mathrm{Ind}_{l,A(l)} & : & A([]) \to (\forall a, l.A(l) \to A(a :: l)) \to \forall l^{\mathbf{L}(\rho)}.A(l)
\end{array}$$

Finally we use the constant derivation term ($\mathsf{IF}_A$),

$$\mathsf{IF}_A : \forall p^{\mathbf{B}}(p \to A) \to ((p \to \bot) \to A) \to A$$

to perform *case distinction* on boolean terms w.r.t. a formula $A$.

*Proof Abbreviations*:
For simplicity, we will use the following proof abbreviations:

$$\dfrac{\exists^+_{x,A} \qquad t \qquad \begin{array}{c}|M \\ A[x/t]\end{array}}{\exists x A}\,\exists^+$$

for

$$\dfrac{\dfrac{\exists^+_{x,A} \qquad t}{A[x/t] \to \exists x A} \to^- \qquad \begin{array}{c}|M \\ A[x/t]\end{array}}{\exists x A} \to^-$$

and

$$\dfrac{\exists^-_{x,A,C} \qquad \begin{array}{c}|M \\ \exists x A\end{array} \qquad \begin{array}{c}|N \\ \forall x(A \to C)\end{array}}{C}\,\exists^-$$

for

$$\dfrac{\dfrac{\exists^-_{x,A,C} \qquad \exists x A \qquad \begin{array}{c}|M\\ \forall x(A \to C) \to C\end{array}}{\forall x(A \to C) \to C}\to^- \qquad \begin{array}{c}|N\\ \forall x(A \to C)\end{array}}{C}\to^-$$

Given a goal formula $C$, the application of the *cases* proof tactic on $t$ generate the following proof tree:

$$\dfrac{\dfrac{\dfrac{\mathsf{IF}_C\ t}{(t \to C) \to ((t \to \mathsf{ff}) \to C) \to C} \qquad \begin{array}{c}|M\\ t \to C\end{array}}{((t \to \mathsf{ff}) \to C) \to C} \qquad \begin{array}{c}|N\\ (t \to \mathsf{ff}) \to C\end{array}}{C}$$

that we will simply rewrite as:

$$\dfrac{\mathsf{IF}_C\ t \qquad \begin{array}{c}|M\\ t \to C\end{array} \qquad \begin{array}{c}|N\\ (t \to \mathsf{ff}) \to C\end{array}}{C}\ (\mathsf{if})$$

We simulate $\vee$-introduction by

$$\dfrac{\exists^+_{x,C} \qquad \mathsf{tt} \qquad \begin{array}{c}|M\\ C[x/\mathsf{tt}]\end{array}}{A \vee B}\vee_0^+$$

with $C[x/\mathsf{tt}] \equiv (\mathsf{tt} \to A) \wedge ((\mathsf{tt} \to \mathsf{ff}) \to B)$, and

$$\dfrac{\exists^+_{x,C} \qquad \mathsf{ff} \qquad \begin{array}{c}|M\\ C[x/\mathsf{ff}]\end{array}}{A \vee B}\vee_1^+$$

with $C[x/\mathsf{ff}] \equiv (\mathsf{ff} \to A) \wedge ((\mathsf{ff} \to \mathsf{ff}) \to B)$. Finally, by $(\mathsf{if})$ we can mimic the $\vee$-elimination as follow:

$$\dfrac{\dfrac{\mathsf{IF} \quad p \qquad \dfrac{\dfrac{(p \to A) \wedge ((p \to \mathsf{ff}) \to B)}{p \to A} \qquad p}{A} \qquad \begin{array}{c}|M\\ \dfrac{C}{p \to C}\end{array} \qquad \dfrac{\dfrac{(p \to A) \wedge ((p \to \mathsf{ff}) \to B)}{((p \to \mathsf{ff}) \to B)} \qquad p \to \mathsf{ff}}{B} \qquad \begin{array}{c}|N\\ \dfrac{C}{(p \to \mathsf{ff}) \to C}\end{array}}{C}}{((p \to A) \wedge ((p \to \mathsf{ff}) \to B)) \to C}}{\forall p((p \to A) \wedge ((p \to \mathsf{ff}) \to B)) \to C}$$

$$\dfrac{\exists^-_{p,D,C} \qquad \begin{array}{c}|\Sigma\\ A\vee B\end{array} \qquad \Big\downarrow}{C}$$

will be shortly rewritten by

$$\dfrac{\exists^-_{p,D,C} \qquad \begin{array}{c}|\Sigma\\ A\vee B\end{array} \qquad \begin{array}{cc}|R & |S\\ A & B\\ |M & |N\\ C & C\end{array} \qquad \begin{array}{c}|K\\ \forall pD\to C\end{array}}{C}\ \vee^-$$

with $D \equiv (p \to A) \wedge ((p \to \mathsf{ff}) \to B)$.

## 2.1.3 Normalization of Proofs

A derivation in normal form does not make "detours", or more precisely, it cannot occur that an elimination rule immediately follows an introduction rule. We now spell out in detail which conversions we shall allow: this is done for derivations written in tree notation and also as derivation terms.

### 2.1.3.1 Conversions

*$\wedge$-conversions*

$$\dfrac{\dfrac{\begin{array}{c}|M\\ A\end{array} \quad \begin{array}{c}|N\\ B\end{array}}{A\wedge B}\wedge^+}{A}\wedge^-_0 \qquad \longmapsto \qquad \begin{array}{c}|M\\ A\end{array}$$

or written as a lambda-term

$$\pi_0(\langle M^A, N^B\rangle) \longmapsto M^A$$

$$\dfrac{\dfrac{\begin{array}{c}|M\\ A\end{array} \quad \begin{array}{c}|N\\ B\end{array}}{A\wedge B}\wedge^+}{B}\wedge^-_1 \qquad \longmapsto \qquad \begin{array}{c}|N\\ B\end{array}$$

or written as a lambda-term

$$\pi_1(\langle M^A, N^B\rangle) \longmapsto N^B$$

$\rightarrow$-*conversion*

$$\dfrac{\dfrac{\begin{array}{c}|M\\B\end{array}}{A \to B}\,{\to}^{+}_{u} \qquad \begin{array}{c}|N\\A\end{array}}{B}\,{\to}^{-} \quad \longmapsto \quad \begin{array}{c}|N\\A\\|M\\B\end{array}$$

or written as a derivation term

$$(\lambda u^A M^B) N^A \longmapsto M[u^A/N^A]$$

$\forall$-*conversion*

$$\dfrac{\dfrac{\begin{array}{c}|M\\A\end{array}}{\forall x A}\,\forall^{+} \qquad t}{A[x/t]}\,\forall^{-} \quad \longmapsto \quad \begin{array}{c}|M'\\A[x/t]\end{array}$$

or written as a derivation term

$$(\lambda x M^A)^{\forall x A} t \longmapsto M^A[x/t]$$

$\forall^{\mathsf{nc}}$-*conversion*

$$\dfrac{\dfrac{\begin{array}{c}|M\\A\end{array}}{\forall x A}\,\forall^{\mathsf{nc}+} \qquad t}{A[x/t]}\,\forall^{\mathsf{nc}-} \quad \longmapsto \quad \begin{array}{c}|M'\\A[x/t]\end{array}$$

or written as a derivation term

$$(\lambda^{\mathsf{nc}} x M^A)^{\forall^{\mathsf{nc}} x A} t \longmapsto M^A[x/t]$$

#### 2.1.3.2 Strong Normalization

No matter in which order we apply the conversion rules, they will always terminate and produce a derivation in "normal form", where no further conversions can be applied.

**Theorem 2.1.1** ([36]). *Every proof-term is strongly normalizing, that is every reduction sequence starting from a proof term $M$, terminates.*

### 2.1.4 Short Excursus in Program Extraction from Proofs

Clearly proper existence proofs have computational content. A well-known and natural way to define this concept is the notion of realizability, which can be

seen as an incarnation of the Brouwer-Heyting-Kolmogorov interpretation of proofs.

#### 2.1.4.1 Type of a Formula

We indicate by $\tau(A)$ as the type of the term (or "program") to be extracted from a proof of $A$. More precisely, to every formula $A$ it is possible to assign an object $\tau(A)$ (a type or the "nulltype" symbol $\varepsilon$). In case $\tau(A) = \varepsilon$ proofs of $A$ have no computational content; such formulas $A$ are called *Harrop formulas*.

$$\begin{aligned}
\tau(P(\vec{x})) &= \begin{cases} \alpha_P & \text{if } P \text{ is a predicate variable with assigned } \alpha_P \\ \varepsilon & \text{Otherwise} \end{cases} \\
\tau(\exists x^\rho A) &= \begin{cases} \rho & \text{if } \tau(A) = \varepsilon \\ \rho \times \tau(A) & \text{Otherwise} \end{cases} \\
\tau(\forall x^\rho A) &= \begin{cases} \varepsilon & \text{if } \tau(A) = \varepsilon \\ \rho \to \tau(A) & \text{Otherwise} \end{cases} \\
\tau(\exists^{\mathsf{nc}} x^\rho A) &= \tau(A) \\
\tau(\forall^{\mathsf{nc}} x^\rho A) &= \tau(A) \\
\tau(A \wedge B) &= \begin{cases} \tau(A) & \text{if } \tau(B) = \varepsilon \\ \tau(B) & \text{if } \tau(A) = \varepsilon \\ \tau(A) \times \tau(B) & \text{Otherwise} \end{cases} \\
\tau(A \to B) &= \begin{cases} \tau(B) & \text{if } \tau(A) = \varepsilon \\ \varepsilon & \text{if } \tau(B) = \varepsilon \\ \tau(A) \to \tau(B) & \text{Otherwise} \end{cases}
\end{aligned}$$

#### 2.1.4.2 Extraction Map

From every derivation $M$ of a computationally meaningful formula $A$ (that is, $\tau(A) \neq \varepsilon$) it is possible to define its *extracted program* $[\![M]\!]$ of type $\tau(A)$[24]. If $\tau(A) = \varepsilon$ then $[\![M]\!] = \varepsilon$.

$$\begin{aligned}
[\![u^A]\!] &= x_u^A \quad (x_u^A \text{ uniquely associated with } A) \\
[\![\lambda u^A M]\!] &= \begin{cases} [\![M]\!] & \text{if } \tau(A) = \varepsilon \\ \lambda x_u^{\tau(A)} [\![M]\!] & \text{Otherwise} \end{cases} \\
[\![M^{A \to B} N^B]\!] &= \begin{cases} [\![M]\!] & \text{if } \tau(A) = \varepsilon \\ [\![M]\!][\![N]\!] & \text{Otherwise} \end{cases} \\
[\![\langle M^A, N^B \rangle]\!] &= \begin{cases} [\![N]\!] & \text{if } \tau(A) = \varepsilon \\ [\![M]\!] & \text{if } \tau(B) = \varepsilon \\ \langle [\![M]\!], [\![N]\!] \rangle & \text{Otherwise} \end{cases}
\end{aligned}$$

$$
\begin{array}{rcl}
[\![M^{A\wedge B} i]\!] & = & \begin{cases} [\![M]\!] & \text{if } \tau(A)=\varepsilon \text{ or } \tau(B)=\varepsilon \\ \pi_i [\![M]\!] & \text{if Otherwise} \end{cases} \\
[\![(\lambda x^\rho M)^{\forall x A}]\!] & = & \lambda x^\rho [\![M]\!] \\
[\![M^{\forall x A} t]\!] & = & [\![M]\!] t \\
[\![(\lambda x^\rho M)^{\forall^{\mathsf{nc}} x A}]\!] & = & [\![M]\!] \\
[\![M^{\forall^{\mathsf{nc}} x A} t]\!] & = & [\![M]\!]
\end{array}
$$

Content of the proof constants:

$$
\begin{array}{rcl}
[\![\exists^-_{x^\rho,A,B}]\!] & = & \begin{cases} \lambda x^\rho f^{\rho\to\tau(B)}.fx & \text{If } \tau(A)=\varepsilon \\ \lambda x^{\rho\times\tau(A)} f^{\rho\to\tau(A)\to\tau(B)}.f(\pi_0 x)(\pi_1 x) & \text{Otherwise} \end{cases} \\
[\![\exists^+_{x^\rho,A}]\!] & = & \begin{cases} \lambda x^\rho x & \text{If } \tau(A)=\varepsilon \\ \lambda x^\rho y^{\tau(A)}.\langle x,y\rangle & \text{Otherwise} \end{cases} \\
[\![(\exists^{\mathsf{nc}})^-_{x^\rho,A,B}]\!] & = & \begin{cases} \lambda x^{\tau(B)}.x & \text{If } \tau(A)=\varepsilon \\ \lambda x^{\tau(A)} f^{\tau(A)\to\tau(B)}.fx & \text{Otherwise} \end{cases} \\
[\![(\exists^{\mathsf{nc}})^+_{x^\rho,A}]\!] & = & \lambda x^{\tau(A)} x \\
[\![\mathsf{IF}_A]\!] & = & \lambda b^{\mathbf{B}}, l^{\tau(A)}, r^{\tau(A)}.(\mathsf{if}\, b\, l\, r) \quad \text{If } \tau(A)\neq\varepsilon \\
[\![\mathrm{Ind}_{n,A(n)}]\!] & = & \mathcal{R}^\sigma_{\mathbf{N}} \\
[\![\mathrm{Ind}_{l,A(l)}]\!] & = & \mathcal{R}^\sigma_{\mathbf{L}(\rho)} \\
[\![\mathrm{Ind}_{t,A(t)}]\!] & = & \mathcal{R}^\sigma_{\mathbf{B}}
\end{array}
$$

#### 2.1.4.3 Realize a Formula

Correctness of the extracted programs is guaranteed by the notion of *modified realizability*. Intuitively, if $t$ is the extracted program from the derivation $M$ of the formula $A$ equal to $\forall x \exists y.P(x,y)$ then for each $x$ the formula $P(x,t(x))$ is provable correct (*Soundness*) i.e. $t$ *(modified) realize* $A$ (written $(t\, \boldsymbol{mr}\, A)$)

$$
\begin{array}{rcl}
r\,\mathbf{mr}\,P(\vec{t}) & = & P(\vec{t}) \\
r\,\mathbf{mr}\,(\exists x.A) & = & \begin{cases} \varepsilon\,\mathbf{mr}\,A[x/r] & \text{if } \tau(A)=\varepsilon \\ \pi_1 r\,\mathbf{mr}\,A[x/\pi_0 r] & \textit{Otherwise} \end{cases} \\
r\,\mathbf{mr}\,(\forall x.A) & = & \begin{cases} \forall x.\varepsilon\,\mathbf{mr}\,A & \text{if } \tau(A)=\varepsilon \\ \forall x.rx\,\mathbf{mr}\,A & \textit{Otherwise} \end{cases} \\
r\,\mathbf{mr}\,(\exists^{\mathsf{nc}} x.A) & = & \begin{cases} \exists^{\mathsf{nc}} x.\varepsilon\,\mathbf{mr}A & \text{if } \tau(A)=\varepsilon \\ \exists^{\mathsf{nc}} x.r\,\mathbf{mr}\,A & \textit{Otherwise} \end{cases}
\end{array}
$$

$$r\,\mathbf{mr}\,(\forall x^{\mathsf{nc}}.A) \;=\; \begin{cases} \forall^{\mathsf{nc}} x.\varepsilon\,\mathbf{mr}\,A & \text{if } \tau(A)=\varepsilon \\ \forall^{\mathsf{nc}} x.r\,\mathbf{mr}\,A & \textit{Otherwise} \end{cases}$$

$$r\,\mathbf{mr}\,(A\to B) \;=\; \begin{cases} \varepsilon\,\mathbf{mr}\,A\to r\,\mathbf{mr}\,B & \text{if } \tau(A)=\varepsilon \\ \forall x.x\,\mathbf{mr}\,A\to \varepsilon\,\mathbf{mr}\,B & \text{if } \tau(A)\neq\varepsilon=\tau(B) \\ \forall x.x\,\mathbf{mr}\,A\to rx\,\mathbf{mr}\,B & \textit{Otherwise} \end{cases}$$

$$r\,\mathbf{mr}\,(A\wedge B) \;=\; \begin{cases} \varepsilon\,\mathbf{mr}\,A\wedge r\,\mathbf{mr}\,B & \text{if } \tau(A)=\varepsilon \\ r\,\mathbf{mr}\,A\to \varepsilon\,\mathbf{mr}\,B & \text{if } \tau(B)=\varepsilon \\ \pi_0 r\,\mathbf{mr}\,A\to \pi_1 r\,\mathbf{mr}\,B & \textit{Otherwise} \end{cases}$$

**Theorem 2.1.2** (Soundness)**.** *Let $M$ be a derivation of a formula $A$ from assumptions $u_i : A_i$. Then we can find a derivation of the formula $([\![M]\!]\ \boldsymbol{mr}\ A)$ from assumptions $\bar{u}_i\colon x_{u_i}\ \boldsymbol{mr}\ A_i$.*

*Proof.* By structural induction on $M$ ([36]). □

## 2.2 A First Example of Proof Transformation: How to Extract Programs with *let*

In a proof it can happen that, to prove $B$, we need to prove an auxiliary formula $A$:

$$\dfrac{\dfrac{\begin{array}{c}|M\\ B\end{array}}{A\to B}\qquad \begin{array}{c}|N\\ A\end{array}}{B}$$

This create a detour that, once normalized, reduce to

$$\begin{array}{c}|N\\ A\\ |M\\ B\end{array}$$

That is $|N$, with end formula $A$, is substituted for all the open assumptions $u^A$ in $M$. At programming level this conversion is represented by following $\beta$-reduction:

$$(\lambda\, x^{\tau(A)}[\![M]\!]^{\tau(B)})[\![N]\!]^{\tau(A)} \longrightarrow_\beta [\![M]\!]^{\tau(B)}[x^{\tau(A)}/[\![N]\!]^{\tau(A)}]$$

with $\tau(A),\tau(B)\neq\epsilon$. Clearly the piece of code $[\![N]\!]^{\tau(A)}$ will be duplicated as many times $x^{\tau(A)}$ appear free in $[\![M]\!]^{\tau(B)}$. A way to create more compact code is replace the original proof by:

$$\dfrac{\dfrac{\mathsf{Id} : (A \to B) \to A \to B \qquad \dfrac{\begin{array}{c}|M\\ B\end{array}}{A \to B}}{A \to B} \qquad \begin{array}{c}|N\\ A\end{array}}{B}$$

With $\mathsf{Id}$ the *identity axiom*. If $\mathsf{Id}$ is *not animated*[37], then it is considered as a *back-box* proof-term and is not involved in any simplification. The content of the previous proof is:

$$(\mathsf{Id}^{\tau((A\to B)\to A\to B)}\ \lambda\, x^{\tau(A)} [\![M]\!]^{\tau(B)})\, [\![N]\!]^{\tau(A)}$$

If we consider a *call-by-value* evaluation strategy the argument of the application is evaluated first, and the previous program is printed as

$$\mathsf{let}\ x\ [\![N]\!]\ [\![M]\!]$$

with the obvious meaning: set $x$ equal to $[\![N]\!]$, then execute $[\![M]\!]$. An interesting application of this program replacement is in the context of the proofs by induction. Consider the derivation:

$$\dfrac{\mathrm{Ind}_{n,\forall x^\rho A(n)} \qquad \begin{array}{c}|N\\ \forall x^\rho A(0)\end{array} \qquad \dfrac{\begin{array}{c}|M\\ \forall x^\rho A(n+1)\end{array}}{\forall x^\rho A(n) \to \forall x^\rho A(n+1)}}{\forall n(\forall x^\rho A(n) \to \forall x^\rho A(n+1))}}{\forall n \forall x^\rho A(n)}$$

Assuming $\tau(A) \neq \epsilon$, the algorithmic content of the step case is:

$$\alpha \equiv \lambda n \lambda x^{\rho \to \tau(A(n))} [\![M]\!]^{\rho \to \tau(A(n+1))}$$

Now suppose $x$ appear several times inside $[\![M]\!]$ and each time in the applicative form $(x\, t^\rho)$, for some $t$. This will produce severals executions of same code when the term $\alpha$ is applied to a natural number and to a functional term. To avoid this phenomena we substitute the proof in step case $M$ by:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{Id} : \sigma \qquad \dfrac{\begin{array}{c}|M\\ \forall x^\rho A(n+1)\end{array}}{(A(n)[x^\rho/t^\rho] \to \forall x^\rho A(n+1))}}{A(n)[x^\rho/t^\rho] \to \forall x^\rho A(n+1)} \qquad \dfrac{[u : \forall x^\rho A(n)]\ \ t^\rho}{A(n)[x^\rho/t^\rho]}}{\forall x^\rho A(n+1)}}{\forall x^\rho A(n) \to \forall x^\rho A(n+1)}}{\forall n(\forall x^\rho A(n) \to \forall x^\rho A(n+1))}$$

with $\sigma \equiv (A(n)[x^\rho/t^\rho] \to \forall x^\rho A(n+1)) \to A(n)[x^\rho/t^\rho] \to \forall x^\rho A(n+1)$. The computational content of the modified step case is:

$$\lambda n, x^{\rho\to\tau(A(n))}(\mathsf{Id}^{\tau(\sigma)}\ \lambda x^{\tau(A(n))}[\![M]\!]^{\rho\to\tau(A(n+1))})\,(x^{\rho\to\tau(A(n))}\ t)^{\tau(A(n))}$$

that is printed as

$$\lambda n, x\,|\mathsf{let}\ y\ (x\ t)\ [\![M]\!]$$

that is, given a natural and a real procedure $f$ (the recursive call), $f$ is applied on $t$, the returning value binded by $y$ and $[\![M]\!]$ (where $y$ may occur) executed.

## 2.3 Minlog

Minlog is intended to reason about computable functionals, using minimal logic. It is an interactive prover with the following features [36]:

- Proofs are treated as first class objects: they can be normalized and then used for reading off an instance if the proven formula is existential, or changed for program development by proof transformation.
- To keep control over the complexity of extracted programs, we follow Kreisel's proposal and aim at a theory with a strong language and weak existence axioms. It should be conservative over (a fragment of) arithmetic.
- Minlog is based on minimal rather than classical or intuitionistic logic. This more general setting makes it possible to implement program extraction from classical proofs, via a refined $A$-translation (cf. [6]).
- Constants are intended to denote computable functionals. Since their (mathematically correct) domains are the Scott-Ershov partial continuous functionals, this is the intended range of the quantifiers.
- Variables carry (simple) types, with free algebras as base types. The latter need not be finitary (so we allow e.g. countably branching trees), and can be simultaneously generated. Type parameters (ML style) are allowed, but we keep the theory predicative and disallow type quantification. Also predicate variables are allowed, as placeholders for formulas (or more precisely, comprehension terms).
- To simplify equational reasoning, the system identifies terms with the same normal form. A rich collection of rewrite rules is provided, which can be extended by the user. Decidable predicates are implemented via boolean valued functions, hence the rewrite mechanism applies to them as well.

*Notation*:
In the MINLOG proof assistant, extracted programs are presented in a textual style, that we briefly describe now along with the correspondence with the above mathematical notations: in programs produced by MINLOG, tt and ff are typeset `#tt` and `#ff` respectively; $\rho \times \sigma$ as `(rho@@sigma)`, $\mathbf{L}(\rho)$ as `(list rho)`, $\lambda x.t$ is written as `([x]t)`, $(\mathcal{R}^{\sigma}_{\mathbf{N}/\mathbf{B}/\mathbf{L}(\rho)}\, b\, s)$ as `(Rec (nat/bool/list rho => sigma) b s)` and $(\pi_{0/1} e)$ as `(left/right e)`.

# 3 Pruning

## 3.1 Introduction

In this chapter we deal with an old idea first introduced by Christopher Alan Goad in the 1980s[17] called *Pruning*. Pruning is first of all a *proof transformation* to remove redundant (computationally relevant or not) parts of a proof. But pruning is a also a *program transformation*: in the program extracted from a pruned proof redundant chunks of code are dropped making use of a kind of dependency information which does not appear in ordinary programs. For the most part, the redundancies removed by pruning are not to be found in proofs generated by people, however, proofs that result from automatic process tend to include such redundancies. Thus the pruning transformation will not be of much use when applied to proofs of algorithms as originally presented.

The pruning transformation has its theoretical foundation in the work in proof theory of Dag Prawitz.

Dag Prawitz[31] asserts that *redundant* application of ($\vee$E) and ($\exists$E) constitute unnecessary complication in proof, and can be easily removed. A natural deduction proof in normal form *and* without such redundancies is said to be in *full*-normal form . The rules to bring a derivation in *full*-normal form, the *Immediate Simplification* rules [31, pag.254], are depicted in Figure 3.1.

Nine years later Goad showed that the application of the immediate simplification rules (which he called *pruning* rules) to a proof which has been specialized can lead to a very large increase in the efficiency of the extracted algorithm. Pruning has the unusual quality that it modifies the function computed by the expression to which it is applied[17, pp 23,56] while preserving the validity of an algorithm for the specification embodied in the end formula of the proof describing the algorithm.

The pruning *protocol* developed by Goad is based on the following three steps:

**Proof specialization** : specialization of a subset of the input parameters of a given proof.

**Dependency removal transformation** : replacement of all the open assumptions, the type can be derived from a certain *knowledge*, by another proof of the same type. This knowledge will consist in a set of formulas (types of assumption variables) accumulated during a traversal of the proof tree.

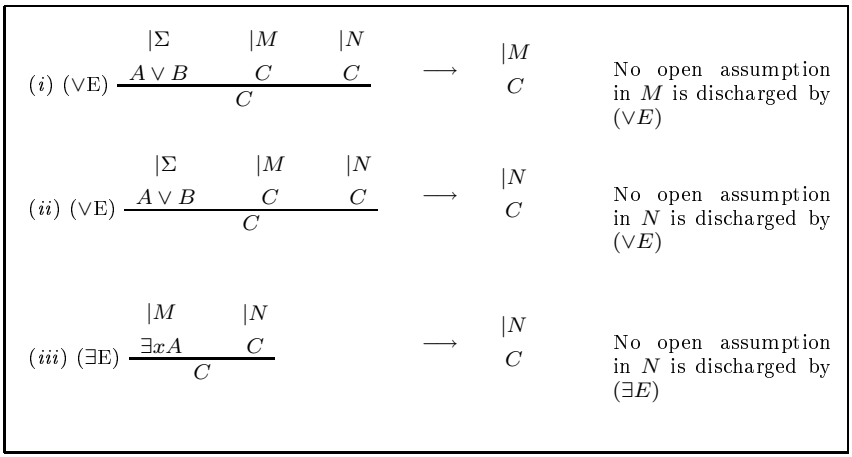



Figure 3.1: Prawitz's *Immediate Simplification* / *Pruning* rules

**Application of the Immediate Simplification /Pruning rules** : simplification of the proof tree with respect to a given set of *pruning* rules in order to eliminate all the $\vee/\exists$ redundant inferences.

In this chapter, we present an implementation of pruning into the Minlog proof assistant. The adaptation is less obvious than what it appears at first view. Several new developments upon the existing work include:

- The demonstration how pruning is intimately related to (and depends on) the operation of *permuting* a proof [41, pag. 180]. Moreover we will show the computational benefits, in terms of elimination of redundant code, that the permutation operation induce on the extracted code.

- The development in Minlog of a proof for the *Bin Packing* problem. After the pruning protocol has been applied on such proof we show the computational benefit on the extracted programs of this operation.

To our knowledge, this is the first implementation of the pruning transformation in a modern proof assistant.

Since Goad's original thesis, the research in this field has expanded in several directions. Berardi[3] and Boerio[7], then later Damiani and Giannini[12] developed a set of techniques in order to eliminate useless code in the programs extracted from proofs. Nogin[29] put a lot of effort in re-implementing many NuPrl tactics in order to make them work more efficiently. Penny Anderson in

her Ph.D. thesis[1] used Frank Pfenning's[30] *lemma insertion* (user dependent) proof transformation in order to extract *tail recursive* programs from proofs. Finally Chiarabini [9] generalized the Anderson's idea producing a completely user-independent proof transformation to obtain the same result.

Before ending this introductory section, in order to show how pruning effects the efficiency of the extracted programs, we present the following,

**Example 3.1.1** (From Goad's thesis [17]). Let $A(x,y,z) \subseteq \mathbf{N} \times \mathbf{N} \times \mathbf{N}$ such that $A(x,y,z) \equiv (x+y \leq z) \wedge (xy \leq z)$. In order to prove that for each pair of naturals $x$ and $y$ there exists $z$ such that $A(x,y,z)$, we define the following axioms:

- $\mathsf{Ax}_1 \equiv \forall x,y((x \leq 1) \rightarrow A(x,y,y+1))$
- $\mathsf{Ax}_2 \equiv \forall x,y((y \leq 1) \rightarrow A(x,y,x+1))$
- $\mathsf{Ax}_3 \equiv \forall x,y((x \leq 1 \rightarrow \bot) \rightarrow (y \leq 1 \rightarrow \bot) \rightarrow A(x,y,2xy))$

Now we can proceed with the following proof $\mathcal{P}_1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{|\Sigma}}{(y \leq 1) \vee (y \not\leq 1)} \quad \dfrac{\dfrac{(\mathsf{Ax}_2\, x\, y) \qquad u_2^{y \leq 1}}{A(x,y,x+1)}}{\exists z A(x,y,z)} \quad \dfrac{\dfrac{(\mathsf{Ax}_3\, x\, y) \qquad v_1^{(x \not\leq 1)}}{(y \not\leq 1) \rightarrow A(x,y,2xy)} \qquad v_2^{(y \not\leq 1)}}{A(x,y,2xy)}}{\exists z A(x,y,z)}}{\exists z A(x,y,z)} \quad \dfrac{\dfrac{\dfrac{}{|\Sigma'}}{(x \leq 1) \vee (x \not\leq 1)} \quad \dfrac{\dfrac{(\mathsf{Ax}_1\, x\, y) \qquad u_1^{x \leq 1}}{A(x,y,y+1)}}{\exists z A(x,y,z)} \quad \exists z A(x,y,z)}{\exists z A(x,y,z)}}{\forall y \exists z A(x,y,z)}}{\forall x,y \exists z A(x,y,z)}$$

Where $\Sigma$, $\Sigma'$ are instantiations of the lemma $\forall x,y(x \leq y) \vee (y \leq x)$ which states the decidability of numerical inequality. The algorithmic content of this proof is the following program $\mathtt{P}_1$:

```
[x,y] [if (x<=1) (y+1)
                 [if (y<=1) (x+1)
                            2xy]]
```

We specialize our proof setting $y$ equal to "1" , that is, we substitute "1" for each free occurrence of $y$ in $\mathcal{P}_1$. The condition $(y \leq 1)$ becomes true, and after normalizing the instantiated proof, the inner case distinction is simplified according to the following proof reduction rules

$$\Sigma^{\mathsf{atom(tt)} \vee B} M^C\ N^C \quad \rightarrow \quad M^C$$

$$\Sigma^{A\vee\mathsf{atom}(\mathtt{tt})}\ M^C\ N^C \quad\to\quad N^C$$

obtaining the following proof $\mathcal{P}_2$:

$$\dfrac{|\Sigma \qquad \dfrac{(\mathsf{Ax}_1\, x\, 1) \qquad u_1^{x\le 1}}{A(x,1,2)}}{\dfrac{(x\le 1)\vee(x\not\le 1) \qquad \exists z A(x,1,z)}{\exists z A(x,1,z)}} \qquad \dfrac{(\mathsf{Ax}_2\, x\, 1) \qquad [1\le 1]}{A(x,1,x+1)}}{\dfrac{\exists z A(x,1,z)}{\forall x \exists z A(x,1,z)}}$$

Such proof correspond to the specialized conditional term $\mathtt{P}_2$:

```
[x] [if (x<=1) 2 (x+1)]
```

The second minor premise of the $(\vee E)$ inference in the specialized proof above does not depend on the assumption $v_1^{(x\not\le 1)}$ and so the rule *ii)* of Table 3.1 applies. We prune $\mathcal{P}_2$ obtaining the following simplified proof $\mathcal{P}_3$:

$$\dfrac{\dfrac{\dfrac{(\mathsf{Ax}_2\, x\, 1) \qquad [1\le 1]}{A(x,1,x+1)}}{\exists z A(x,1,z)}}{\forall x \exists z A(x,1,z)}$$

from which we extract the following lambda abstraction $\mathtt{P}_3$:

```
[x] (x+1)
```

The proofs $\mathcal{P}_2$ and $\mathcal{P}_3$ are different derivations of the same formula $\forall x \exists z A(x,1,z)$ and they have different computational content: in fact meanwhile the program `(P2 0)` rewrites into 2, `(P3 0)` rewrites into 1.

This shows that the application of pruning to a proof can lead to an increase in the efficiency of the extracted algorithm (in this case it consists in discharging the case distinction) and that it modifies the computational behavior of the (computational) content of the proof to which it is applied.

## 3.2 Pruning in Minlog

### 3.2.1 *Immediate Simplification* in Minlog

As we have seen in the previous chapter in our logic we perform case distinction over a boolean term $t$ by the application of the proof constant $\mathsf{IF}_A : \forall b^{\mathbf{B}}((b\to A)\to((b\to\bot)\to A)\to A)$ to $t$. Given a goal formula $A$, the application of the *cases* proof tactic on $t$ generates the following proof tree:

$$\dfrac{\mathsf{IF}_A\ t \qquad \begin{matrix}|M\\ t\to A\end{matrix} \qquad \begin{matrix}|N\\ (t\to\bot)\to A\end{matrix}}{A}\ (\mathsf{if})$$

where $M$ and $N$ are the proofs the user will have to supply. The derivation rule (if) could be seen as an $(\vee\exists)$ inference where the or formula to eliminate $A \vee B$ is just $t \vee \neg t$. In order to act on more general formulas than $\mathsf{atom}(t)$ we remember that in our system we adopted the following convention

$$A \vee B := \exists p(p \to A \wedge (p \to \bot) \to B)$$

So, if we need to prove $C$ dispatching over the truth $A$ or $B$ we can proceed building the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{(p \to A) \wedge ((p \to \bot) \to B)}{p \to A} \quad p}{A}}{B}}{\mathsf{IF}\ p \quad p \to B} \qquad \dfrac{\dfrac{\dfrac{\dfrac{(p \to A) \wedge ((p \to \bot) \to B)}{((p \to \bot) \to B)} \quad p \to \bot}{B}}{B}}{(p \to \bot) \to B}$$

$$\dfrac{\dfrac{\dfrac{B}{((p \to A) \wedge ((p \to \bot) \to B)) \to B}}{\forall p((p \to A) \wedge ((p \to \bot) \to B)) \to B}}{\exists^- \quad A \vee B \qquad\qquad B}$$

Clearly here the assumption $p$ has to be read as "$A$ holds" meanwhile $p \to \bot$ as "$B$ holds". For this reason we adapted the pruning rules à la Goad (Figure 3.1) to work on (if)-inference patterns rather than general $(\vee\exists)$-inferences, as depicted in Figure (3.2). We can write such rules also as conversion rules between proof-terms as follow:

$$\begin{array}{rrcll}
(\mathsf{IF}\ t & \lambda u^t.M^C & \lambda u^{t\to\bot}.N^C) & \longrightarrow & M^C \qquad u^t \notin \mathrm{FV}(M^C) \\
(\mathsf{IF}\ t & \lambda u^t.M^C & \lambda u^{t\to\bot}.N^C) & \longrightarrow & N^C \qquad u^{t\to\bot} \notin \mathrm{FV}(N^C) \\
(\exists^-_{x,A,C} & M^{\exists x A} & \lambda x, u^A.N^C) & \longrightarrow & N^C \qquad u^A \notin \mathrm{FV}(N^C)
\end{array}$$

### 3.2.2 Dependencies Removal Transformation

The dependencies removal transformation improves the effectiveness of pruning. This operation involves the replacement of occurrences of assumption variables, when possible, by proofs of those assumptions from other available information. Consider for example the proof in Figure 3.3.

In $M$ both the assumptions $u_3^{x\le 1}$ and $u_1^{x\le 2}$ are *active*, i.e. they can appear free in $M$ in order to prove $C$. On the other hand, we note that the type of the assumption $u_1$ is logically implied by the type of $u_3$. So we can create the

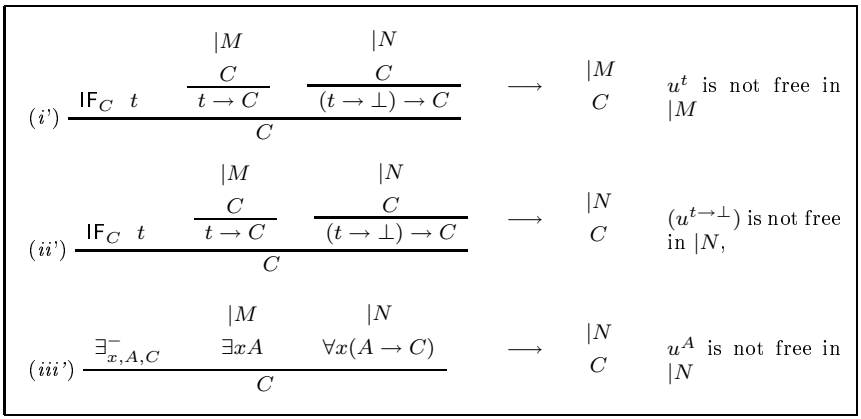



Figure 3.2: Pruning rules for minimal logic.

new proof

$$(\mathsf{AX} : \forall x(x \leq 1 \rightarrow x \leq 2))\, x\, u_3^{x \leq 1}$$

of type $(x \leq 2)$ and substitute it for each open assumption $u_1^{x \leq 2}$ in $M$ (with AX new axiom)

In general we will have to face the following problem: given a conditional proof-term if-cmd of the form

$$(\mathsf{IF}\ t\ \lambda u^t M^C\ \lambda v^{t \rightarrow \bot} N^C)$$

and a *knowledge* KWN (list of axioms, assumption variables, ...) how to simplify if-cmd with respect to KWN? In order to solve this problem, we implemented a procedure named drt(d*ependency* r*emoval* t*ransformation*) shown in Algorithm 1. In the present work we assume KWN to be a list of pairs $(t, u^t)$ with $t$ linear inequality (in the sense that $t$ involves an inequality in some linear function of the variables) and $u^t$ assumption variable of type $t$, assumed during the proof tree traversal plus, (eventually) some *external* knowledge supplied directly by the user.

In Algorithm 1 the truth of the formulas $(\mathsf{KNW} \succ t)$, to be red as 'from the knowledge KNW it is possible to deduce the formula $t$', is decided by a procedure call to the *Simplex Algorithm* (we implemented in Minlog the simplex algorithm reported in [38]).

A final remark regarding the termination of the procedure drt. Given an input proof $p$ the computation of drt is driven by the inductive structure of $p$. If $p$ is a basic proof (assumption variable or proof constant) then drt stops (line

**Algorithm 5** $\mathsf{drt}(p, \mathsf{KNW} = ((t_n, u_n^{t_n}) \dots (t_1, u_1^{t_1})))$, for some $0 \leq n$, $p$ input proof, $t_i$ linear inequality and $u^{t_i}$ assumption variable associated with $t_i$. We indicate by $M[u^\alpha/N^\alpha]$ the substitution in M of all free occurrences of the open assumption $u$ with $N$. We write $\mathsf{AX}^{\forall \vec{x} t_1 \to \dots \to t_n}$ for the axiom $\mathsf{AX}$ of type $\forall \vec{x}(t_1 \to \dots \to t_n)$ with $\vec{x}$ list of variables that occur in $t_1, \dots, t_n$. Given a linear inequality $t$, we indicate with $(\mathsf{KNW} \succ t)$ a boolean condition that holds if and only if $t_1 \to \dots \to t_n \to t$ holds (eventually $n = 0$). Finally proof_constr is a generic proof constructor.

1: **if** $p$ is a proof-constant, axiom or assumption variable **then**
2: $\quad p$
3: **else if** $p \equiv \mathsf{IF}\, t\ (\lambda u^t M)\ (\lambda v^{(t \to \bot)} N)$ **then**
4: $\quad$ **let**
5: $\quad\quad M' =$
6: $\quad\quad$ **if** $(\mathsf{KNW} \succ t)$ **then**
7: $\quad\quad\quad$ **let** $(M'' = \mathsf{drt}(M, \mathsf{KNW}))$ **in** $M''[u^t/(\mathsf{AX}^{\forall \vec{x} t_1 \to \dots \to t_n \to t}\, \vec{x}\, u^{t_1} \dots u^{t_n})]$
8: $\quad\quad$ **else**
9: $\quad\quad\quad \mathsf{drt}(M, ((t, u^t) :: \mathsf{KNW}))$
10: $\quad\quad$ **end if**
11:
12: $\quad\quad N' =$
13: $\quad\quad$ **if** $(\mathsf{KNW} \succ (t \to \bot))$ **then**
14: $\quad\quad\quad$ **let** $(N'' = \mathsf{drt}(N, \mathsf{KNW}))$ **in** $N''[v^{t \to \bot}/(\mathsf{AX}^{\forall \vec{x} t_1 \to \dots \to t_n \to (t \to \bot)}\, \vec{x}\, u^{t_1} \dots u^{t_n})]$
15: $\quad\quad$ **else**
16: $\quad\quad\quad \mathsf{drt}(N, ((t \to \mathsf{ff}), v^{(t \to \bot)}) :: \mathsf{KNW}))$
17: $\quad\quad$ **end if**
18: $\quad$ **in** $(\mathsf{IF}\, t\ (\lambda u^t M')\ (\lambda v^{(t \to \bot)} N'))$
19: **else** {that is $p = (\mathsf{proof\_constr}\ R_1 \dots R_n)$}
20: $\quad$ **let** $R_1' = \mathsf{drt}\ (R_1, \mathsf{KNW}), \dots, R_n' = \mathsf{drt}\ (R_n, \mathsf{KNW})$ **in** $(\mathsf{proof\_constr}\ R_1' \dots R_n')$
21: **end if**

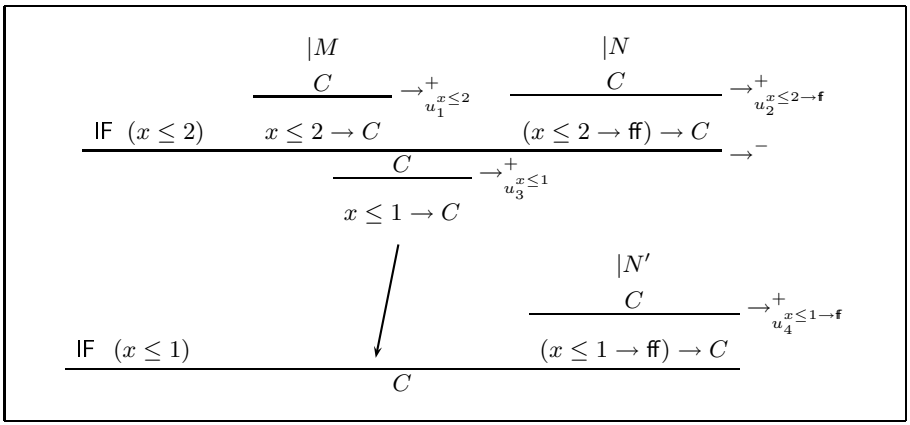



Figure 3.3:

1,2 Algorithm 1). Otherwise, all the recursive calls in drt (lines 7, 9 14, 16, 20 in Algorithm 1) are performed on structurally simpler proof than the input proof $p$.

### 3.2.3 Computing with Permutative Conversions

It is not always possible to perform the *dependencies removal transformation* step of the pruning protocol. The occurrences of particular proof-patterns (example below) make such transformation impossible. Consider the following proof $\mathcal{P}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{t}\ \mathsf{IF} \quad \dfrac{\dfrac{}{\exists x A}}{t \to \exists x A} \quad \dfrac{\dfrac{}{\exists x A}}{(t \to \bot) \to \exists x A}}{\exists x A} \quad \dfrac{\dfrac{}{x}\ \mathsf{IF} \quad \dfrac{\dfrac{}{P}}{t \to C} \quad \dfrac{\dfrac{}{Q}}{(t \to \bot) \to C}}{C}}{\forall x (A \to C)}\ \exists^{-}_{x,A,C}}{C}$$

For space reasons, we indicate the application of the existential elimination axiom just by a label on the right hand side of the last inference, and we dropped the labels $\to^{+}$ associated with the assumption-introduction inferences. The problems that can arise from these kind of proof patterns are essentially two: *i*) the condition $x$ is not comparable with any other boolean condition *ii*)

from such proof-patterns it may be possible to extract code with redundancies that are difficult to eliminate.

For example if we assume the proofs $M, N, P$ and $Q$ to be in normal form, then the entire derivation $\mathcal{P}$ is in normal form. On the other hand, assuming $A$ and $C$ *not* an harrop-formulas, the algorithmic content of $\mathcal{P}$ is the redex:

$$\texttt{([x,q] (if x [[P]] [[Q]]))}\quad \texttt{left(if t [[M]] [[N]])}\quad \texttt{right(if t [[M]] [[N]])} \tag{3.1}$$

Now consider the following instantiation of (3.1) for generic terms `e1,e2,e3` and `t`:

```
([x,q] (if x (#tt, q) (#ff, e1)))
    left (if t<=2 (#tt, e2) (#ff, e3))
    right(if t<=2 (#tt, e2) (#ff, e3))
```
(3.2)

Considering the additional conversion rule that map $f$`(if t r s)` to `(if t` $f$`r` $f$`s)`, `(if t #tt #ff)` to `t`, and `(a,(if t b s))` to `(if t (a,b) (a,s))` then (3.2) reduces to:

```
(if t<=2 (if t<=2 (#tt, e2) (#tt, e3))(#ff, e1))
```
(3.3)

The two nested and redundant `if`'s on the condition `(t<=2)` in the term above have no counterpart at proof level, i.e. in $\mathcal{P}$ we don't find two nested case distinctions on the same condition `(t<=2)` as we could guess looking at the program (3.3). Moreover, the two nested and redundant case distinctions in (3.3) are a source of inefficiency. In order to overcome these problems, we implemented in MINLOG the permutative conversion rule (in the proof-tree style) in Figure 3.4 or written as a conversion rule between proof-terms:

$$\begin{gathered}\alpha \equiv \exists^-(\mathsf{IF}\, t\ \lambda u^t M^{\exists x A}\ \ \lambda u^{t\to\mathsf{ff}} N^{\exists x A})\, Z^{\forall x A\to C} \\ \Longrightarrow \\ \mathsf{IF}\, t\ (\lambda u^t \exists^-\ M^{\exists x A}\, Z^{\forall x A\to C})\ (\lambda u^{t\to\mathsf{ff}} \exists^-\ N^{\exists x A}\, Z^{\forall x A\to C})\end{gathered} \tag{3.4}$$

This rule permutes an existential elimination inference upwards over the minor premises of a case distinction proof (for more details refer to [41, pp, 180]). We see now how particular instances of the conversion rule (3.4) help us in simplifying proof pattern as $\mathcal{P}$ and solve the problems raised in points *i*) and *ii*) above.

Let consider the following specialization in $\alpha$: assume $M^{\exists x A}$ to be the proof term $(\exists^+ \overline{t}\, R^A)^{\exists x A}$. We can rewrite $\alpha$ as:

$$\exists^-(\mathsf{IF}\, t\ \lambda u^t(\exists^+ \overline{t}\, R^A)^{\exists x A}\ \ \lambda u^{t\to\mathsf{ff}} N^{\exists x A})\, Z^{\forall x A\to C}$$

By (3.4) it is converted to:

$$\mathsf{IF}\ t\ (\lambda u^t \exists^-(\exists^+ \overline{t}\, R^A)^{\exists x A}\, Z^{\forall x A\to C})\ (\lambda u^{t\to\mathsf{ff}} \exists^-\ N^{\exists x A}\, Z^{\forall x A\to C})$$

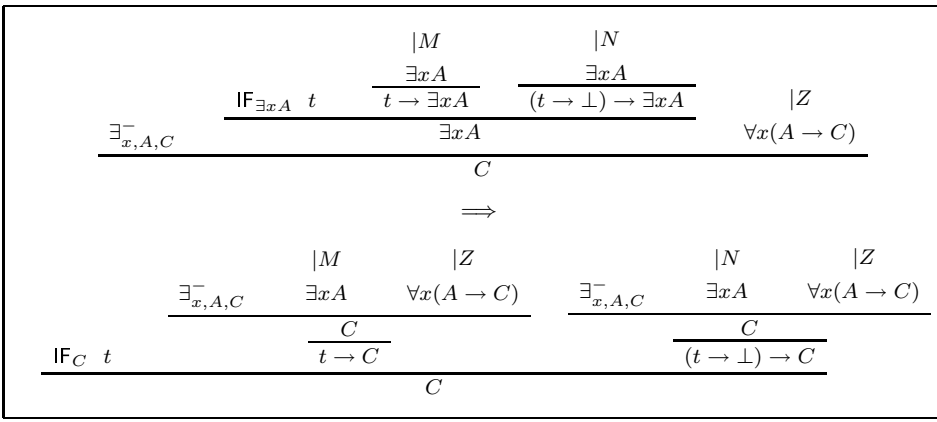


Figure 3.4: Conversion rule to permute an IF followed by an $\exists^-$ axiom.

and eliding the $\exists^-/\exists^+$-axioms by (EIid), we obtain:

$$\alpha' \equiv \mathsf{IF}\ t\ (\lambda u^t Z^{\forall xA \to C}\, \overline{t}\, R^A)\ (\lambda u^{t \to \mathsf{ff}} \exists^-\ N^{\exists xA}\ Z^{\forall xA \to C})$$

If we assume $Z$ be the proof-term $(\lambda x^{\mathbf{B}}, u^A S^C)$ with $S$ case distinction over the condition $x$ (as in $\mathcal{P}$) then a consequence of the instantiation $(Z\,\overline{t})$ in $\alpha'$ is making explicit the term on which the case distinction $S$ is performed, eliminating the problem raised in point $i$) above. We note that just the substitution of $(\exists^+\,\overline{t}\,R)$ for $M$ in $\alpha$ would have no benefit without the permutation rule (3.4). A similar study can be done for the second minor premise of the case distinction in $\alpha'$, for a suitable $N^{\exists xA}$.

Now we see here how the conversion rule (3.4) can help in simplifying redundancies in the extracted code. Let $\xi$ be the following specialization of $\mathcal{P}$ from which the procedure (3.2) could be extracted (here we assume $\tau(C) = \epsilon$):

$$\begin{array}{l} \exists^-\ (\mathsf{if}\ (t \leq 2)\ \lambda u^{(t \leq 2)}\quad (\exists^+\ \mathsf{tt}\ (\exists^+\ \mathsf{e2}\, M^C)^{\exists yC})^{\exists x,yC} \\ \qquad\qquad\qquad\quad \lambda u^{(t \leq 2) \to \bot} (\exists^+\ \mathsf{ff}\ (\exists^+\ \mathsf{e3}\, N^C)^{\exists yC})^{\exists x,yC}) \\ \lambda x, q^{\exists yC} \mathsf{if}\ x\ \lambda u^x\quad (\exists^+\ \mathsf{tt}\ q)^{\exists x,yC} \\ \qquad\qquad\qquad\quad \lambda u^{x \to \bot} (\exists^+\ \mathsf{ff}\ (\exists^+\ \mathsf{e1}\, R^C)^{\exists yC})^{\exists x,yC} \end{array}$$

Now let's permute $\xi$ with (3.4):

$$\begin{array}{l}\mathsf{if}\ (t \leq 2)\ \lambda u^{(t\leq 2)}\,\exists^-\ (\exists^+\ \mathtt{tt}\ (\exists^+\ \mathtt{e2}\ M^C)^{\exists yC})^{\exists x,yC} \\ \qquad\qquad \lambda x, q^{\exists yC}\mathsf{if}\ x\ \lambda u^x\quad (\exists^+\ \mathtt{tt}\ q)^{\exists x,yC} \\ \qquad\qquad\qquad\qquad \lambda u^{x\to\bot}(\exists^+\ \mathtt{ff}\ (\exists^+\ \mathtt{e1}\ R^C)^{\exists yC})^{\exists x,yC} \\ \qquad \lambda u^{(t\leq 2)\to\bot}\,\exists^-\ (\exists^+\ \mathtt{ff}\ (\exists^+\ \mathtt{e3}\ N^C)^{\exists yC})^{\exists x,yC} \\ \qquad\qquad \lambda x, q^{\exists yC}\mathsf{if}\ x\ \lambda u^x\quad (\exists^+\ \mathtt{tt}\ q)^{\exists x,yC} \\ \qquad\qquad\qquad\qquad \lambda u^{x\to\bot}(\exists^+\ \mathtt{ff}\ (\exists^+\ \mathtt{e1}\ R^C)^{\exists yC})^{\exists x,yC}\end{array}$$

Eliminating $\exists^-/\exists^+$ by (Elid) we have:

$$\begin{array}{l}\mathsf{if}\ (t \leq 2)\ \lambda u^{(t\leq 2)}\lambda u^{\mathtt{tt}}\qquad (\exists^+\ \mathtt{tt}\ (\exists^+\ \mathtt{e2}\ M^C)^{\exists yC})^{\exists x,yC} \\ \qquad\qquad \lambda u^{(t\leq 2)\to\bot}\lambda u^{\mathtt{ff}\to\bot}\ (\exists^+\ \mathtt{ff}\ (\exists^+\ \mathtt{e1}\ R^C)^{\exists yC})^{\exists x,yC}\end{array}$$

And finally, extracting the term from the last proof we obtain the simplified code:

```
if t<=2 (#tt, e2) (#ff, e1)
```

## 3.3 Case Study: The Bin Packing Problem

In this section we introduce the 1-dimensional *Bin-Packing* problem, as originally formulated in [17].

Given a list of boxes of dimensions expressed by the naturals $p_1, \ldots, p_n$ and bins of capacity expressed by the naturals $b_1, \ldots, b_m$, find, if it exists, a *valid assignment* of the boxes into the bins in such a way that for each bin the sum of the dimensions of the boxes assigned to it does not exceeds the capacity of the bin itself.

We will indicate the input list of boxes by $X$, the list of bins by $B$, and the output assignment by $A$. We indicate the $i$-th element of $l$, the length of list $l$ and the list $l$ —where the position $i$ is decreased by $a$— respectively by $l[i]$, $|l|$ and $l[i/a]$. The output assignment list $A$ has this property: for each natural $i$, the $i$-th box has to be put in the bin $A[i]$. It follows that the list of boxes and assignments have to have the same length, that is, the equality ($|X| = |A|$) holds. Now we introduce some notation that will be useful for our proof. For lists of naturals $A$ and $X$ and a natural $i$, we define $\mathsf{SUM}$ to be the following function:

$$\mathsf{SUM}(A, X, i) = \sum_{j\in\mathcal{Q}} X[j] \quad \text{with } \mathcal{Q} = \{j | A[j] = i\}$$

We define a predicate $\mathsf{PACK}$ that states under which conditions a list of naturals $A$ can be considered a valid assignment for the list of boxes $X$ and bins $B$, and the additional predicate $\mathsf{PACKB}$ that states the existence of a valid assignment for the list of boxes $X$ and bins $B$ puls an additional constraint on the bin the first box should be associated,

$$\mathsf{PACK}(A, X, B) \iff (\forall i.i < |A| \to A[i] < |B|) \wedge$$

$$(|X| = |A|) \wedge$$
$$(\forall i.i < |B| \rightarrow \mathsf{SUM}(A, X, i) \leq B[i])$$

$$\mathsf{PACKB}(n, A, X, B) \iff X \neq (:) \wedge$$
$$\mathsf{PACK}(A, X, B) \wedge$$
$$(|B| - n) \leq A[0]$$

The 1-dimensional bin packing problem can be formulated as a decision problem where, given an input element $x$, we have to state if there exists an $y$ such that a property $P(x, y)$ holds or not. We can express this fact by the following formula:

$$\forall x(\exists y P(x, y)) \vee ((\exists y P(x, y)) \rightarrow \bot)$$

As already seen, in our system we express such formulas as:

$$\forall x \exists p(p \rightarrow \exists y P(x, y)) \wedge ((p \rightarrow \bot) \rightarrow ((\exists y P(x, y)) \rightarrow \bot))$$

that is, for each input $x$ there exists a boolean $p$ such that if $p$ holds then we are able to supply a solution, else no solution exists. We will call $(p \rightarrow \exists y P(x, y))$ and $((p \rightarrow \bot) \rightarrow ((\exists y P(x, y)) \rightarrow \bot))$ the *positive* and *negative* part of the formula above. The proof-algorithm that we propose is a *first-fit* algorithm because, in the course of the search, it attempts to place a block in the first bin in which it fits as its initial try.

**Theorem 3.3.1.**

$$\forall X, B \exists p \quad (p \rightarrow \exists A \mathit{PACK}(A, X, B)) \wedge$$
$$((p \rightarrow \bot) \rightarrow (\exists A \mathit{PACK}(A, X, B)) \rightarrow \bot)$$

*Proof.* By induction on $X$. Case $X = (:)$. If there are no boxes to fit, then for each list of bins $B$ the empty list is a valid assignment. Case $X = (a :: l)$. Assume the induction hypothesis (IH) and a generic list $B$ of bins. In order to prove

$$\exists p \quad (p \rightarrow \exists A\, \mathsf{PACK}(A, (a :: l), B)) \wedge$$
$$((p \rightarrow \mathsf{ff}) \rightarrow (\exists A \mathsf{PACK}(A, (a :: l), B)) \rightarrow \bot) \tag{3.5}$$

we prove the following assertion:

$$\forall n. \quad (n \leq |B|) \rightarrow \exists p \quad (p \rightarrow \exists A(\mathsf{PACKB}(n, A, (a :: l), B)) \wedge$$
$$((p \rightarrow \bot) \rightarrow (\exists A\, \mathsf{PACKB}(n, A, (a :: l), B)) \rightarrow \bot) \tag{3.6}$$

Obviously we can derive (3.5) instantiating (3.6) on $|B|$. To prove (3.6) we go

by induction on $n$. Case $n = 0$. We fail finding a valid assignment for $(a :: l)$ in $B$ because it should holds $|B| \leq A[0]$ and $A[0] < |B|$. Case $n + 1$. Assume the nested induction hypothesis (NIH) and $(n + 1 \leq |B|)$. We prove:

$$\begin{aligned} \exists p \quad & (p \to \exists A\, \mathsf{PACKB}((n+1), A, (a :: l), B)) \wedge \\ & ((p \to \mathsf{ff}) \to (\exists A\, \mathsf{PACKB}((n+1), A, (a :: l), B)) \to \bot) \end{aligned} \tag{3.7}$$

Obviously if $(n + 1 \leq |B|)$ then $(n \leq |B|)$. There are only two cases:

- $(a \leq B[|B| - (n + 1)])$: The dimension of the first box fits in the bin in position $(|B| - (n + 1))$. So we check if a valid assignment exists for the list $l$ into the list of bins $B$, where the position $(|B| - (n + 1))$ (of $B$) is decreased by the quantity $a$. We instantiate (IH) on $B[(|B| - (n+1))/a]$. So there exists a boolean $\overline{p}$ such that:

$$\begin{aligned} & (\overline{p} \to \exists A\, \mathsf{PACK}(A, l, B[(|B| - (n + 1))/a])) \wedge \\ & ((\overline{p} \to \bot) \to (\exists A\, \mathsf{PACK}(A, l, B[(|B| - (n + 1))/a])) \to \bot) \end{aligned} \tag{3.8}$$

  There are two cases: $\overline{p}$ holds or it doesn't hold.

  $\overline{p}$ **holds**: We are done. From (3.8) we know $\overline{A}$ such that $\mathsf{PACK}(\overline{A}, l, B[(|B| - (n + 1))/a])$, so the thesis is proved introducing $\mathsf{tt}$ for $p$ and $((|B| - (n + 1)) :: \overline{A})$ for $A$ in the positive part of (3.7).

  $(\overline{p} \to \mathsf{ff})$ **holds** : A valid assignment $A$, if it does exists, has to assign $a$ to the bin $i$ with $|B| - n \leq i < |B|$. So the searched assignment and the proof of its existence (or the proof of its non existence) is given by the nested induction hypothesis (NIH)

- $(a \not\leq B[|B| - (n + 1)])$: Also in this case, if a solution does exists, it is given by (NIH)

□

The code extracted from the previous proof is:

```
(Rec (list nat=> list nat=> (boole, list nat)))
 ([B] (#tt , (:)))
 ([a,l,f,B]
   [(Rec (nat => (boole, list nat))
      (#ff, (:))
      [n, (p ,A)]
        if (a <= B[|B|-(n+1)])
          let (p',A') = f B[(|B|-n)/a]
             if p' (#tt, (|B|-(n+1))::A') (p, A)
          (p, A)]  |B|
```

The `let` constructor is obtained using in some strategic point of the proof the identity axiom (Section 2.2).

### 3.3.1 Experiment

We specialize the bin-packing proof on the input lists of boxes $X = (n :: m :)$ and bins $B = (a :: a :)$. The content of the specialized proof is:

```
if (n<=a)
  if (m<=a--n)
   (#tt, 0::(if (m<=a-n) (0:) (if (m<=a) (1:) (:))))
   if (m<=a)
    (#tt, 0::(if (m<=a-n) (0:) (if (m<=a) (1:) (:))))
    if (n<=a)
     if (m<=a)
      (#tt, 1::(if (m<=a) (0:) (if (m<=a-n) (1:) (:))))
      (m<=a-n@
      if (m<=a-n)
        (1::(if (m<=a) (0:) (if (m<=a-n) (1:) (:))))
        (:))
     (#ff, (:))
  if (n<=a)
   if (m<=a)
    (#tt, 1::(if (m<=a) (0:) (if (m<=a-n) (1:) (:))))
    (m<=a-n,
     if (m<=a-n) (1::(if (m<=a) (0:) (if (m<=a-n) (1:) (:))))) (:))
   (#ff, (:))
```

The permutative conversion rules are applied to the specialized proof. The computational content of the permuted proof is:

```
 if (n<=a)
  if (m<=a-n)
   (#tt,  (0::0:))
   if (m<=a)
    (#tt,(0::1:))
    if (n<=a)
     if (m<=a) (#tt, (1::0:)) (m<=a-n, (if (m<=a-n) (1::1:) (:)))
     (#ff, (:))
  if (n<=a)
   if (m<=a) (#tt, (1::0:)) (m<=a-n, (if (m<=a-n) (1::1:) (:)))
  (#ff, (:))
```

Finally pruning is applied to the permuted proof. Here we distinguish two cases in extracting the code from the pruned proof:

**No additional knowledge on $n$,$m$ is required**:

```
P1= if (n<=a)
     if (m<=a-n) (#tt, (0::0:)) (m<=a, (if (m<=a) (0::1:) (:)))
     (#ff, (:))
```

**Assuming $m \leq n$**:

```
P2 = (n<=a, if (n<=a) (0::1:) (:))
```

In the first case (no knowledge on the input parameters) the effect of pruning is the simplification of `if`-statements that occur in the left/right branch of an outer `if`-statement with the same boolean condition. This process could be performed with a program transformation techniques such as *partial evaluation*[21].

In the second case ($m \leq n$) something different happens: there is no way to go from `P1` to `P2` with any program transformation technique. In fact, the elimination of the `if`-statement on the condition `(m<=a-n)` only refers to dependency information available at proof level, and not at program level. Finally we see that the extensional behavior of the extracted code `P1` and `P2` changes. While `P1`, if `n<=a` may return `(0::0:)`, `P2` will always return `(0::1:)`. But pruning keeps the end formula of the sub-proofs on which it is applied so both the results even if different will satisfy the same logical *specification*.

## 3.4 Conclusions

In this chapter we presented an adaptation of the *pruning* technique[17] to minimal logic on which the Minlog proof assistant is based and we applied it to the formalization and simplification of the bin packing problem. In our work we showed how pruning is intimately related to the operation of proof permutation and we showed the computational benefits, in terms of elimination of redundant code, that the permutation operation induce s on the extracted code. In chapter 5 we will propose an extension of the pruning technique.

# 4 Bounded Perfect Matching Problem

## 4.1 Introduction and Motivation

In this section we introduce a widely studied problem in Bioinformatcs, the *shortest common superstring* problem. The problem can be formulated as follows: given a set of strings $\mathcal{P} = \{s_1, \ldots, s_n\}$ find the shortest string $S$ that contains every string in $\mathcal{P}$. For example a superstring of *abc* and *cfa* is *wcfaabcd* but *abcfa* and *cfabc* are the shortests.

The problem of finding the shortest superstring have applications in data compression but the major motivation is related to the sequence assembly problem in shotgun sequencing, a method used for sequencing long DNA strands. Each string in the set $\mathcal{P}$ models one of the sequenced DNA fragments created by the shotgun sequencing protocol[20, pp. 420]. The assembly problem is to deduce the originating DNA string $S$ from the set of sequenced fragments $\mathcal{P}$. Without sequencing errors, the originating string $S$ is a superstring of $\mathcal{P}$ and, under some assumptions, $S$ is likely to be a shortest superstring of $\mathcal{P}$. In that case, a shortest superstring of $\mathcal{P}$ is a good candidate for the originating string $S$.

In [40] it is formally showed that the shortest common superstring is a NP-hard problem, that is there is no polynomial time algorithm solving it (unless P=NP). An idea to solve this problem is to embed it into more familiar algorithmic fields, namely Hamiltonian circuit problems.

Let $s_1, \ldots, s_m$ be a list of strings. We indicate by $o(s_i, s_{i+1})$, $p(s_i, s_{i+1})$ and $s(s_i, s_{i+1})$ the lengths of the overlap, of the prefix and of the suffix between the strings $s_i$ and $s_{i+1}$. Here we use the notion of "prefix" and "overlap" defined as follow. Given decompositions of strings $S = XY$ and $T = YZ$ such that $Y$ is the longest suffix of $S$ (different form $S$) and also a prefix of $T$, we call $Y$, $X$ and $Z$ respectively the the overlap, prefix and suffix strings of $S$ with $T$. These definitions give rise to two graphs, called *overlap graph*, and *prefix graph* for a string list $s_1, \ldots, s_m$. Both are directed graphs that have $m$ nodes labeled $s_1, \ldots, s_m$ and directed edges between any two such nodes (thus also from every node back to itself). Furthermore, the edge pointing from node $s_i$ to $s_j$ is weighted by number $o(s_i, s_j)$ in the overlap graph, and $p(s_i, s_j)$ in the prefix graph.

As showed in [40], searching for a shortest common superstring might as well replaced with searching for the cheapest Hamiltonian cycle (closed path

visiting each node exactly once) through the prefix graph. Unfortunately the Hamiltonian cycle problem is NP-complete, but as it is known from algorithm theory, computing a finite set of disjoint cycles (instead of a single cycle) having minimum summed costs and covering every node in a weighted graph is an efficiently solvable problem. Such a finite set of cycles is called a *cycle cover*.

As proved [40] the problem of computing a *cycle cover with minimum costs in a prefix graph* is equivalent to the problem of computing a *cycle cover with maximum costs in overlap graph*. To do so, we transform the cycle cover problem for overlap graphs into a perfect matching problem in a *bipartite version* of overlap graph. The latter is defined as follows. Create for every node $s_i$ in overlap graph a copy node called $g_i$. Thus the new graph consists of two parts, a left part with all the nodes from the original graph, and a right part that is a copy of the left part. Every directed edge from node $s_i$ to node $s_j$ is simulated by an undirect edge between node $s_i$ and copy node $g_j$ with weight $o(s_i, s_j)$. Now consider an arbitrary local cycle with cost $c$ in overlap graph:

$$s_{\pi_1} \longrightarrow s_{\pi_2} \longrightarrow s_{\pi_3} \longrightarrow \ldots \longrightarrow s_{\pi_{m-1}} \longrightarrow s_{\pi_m} \longrightarrow s_{\pi_1}$$

for some permutation $\pi$. Its directed edges correspond to undirected edges of the bipartite version as follows:

$$\begin{array}{ccccccccc} s_{\pi_1} & & s_{\pi_2} & & s_{\pi_3} & \cdots & s_{\pi_{m-1}} & & s_m \\ | & & | & & | & & | & & | \\ g_{\pi_2} & & g_{\pi_3} & & g_{\pi_4} & \cdots & g_{\pi_m} & & g_{\pi_1} \end{array}$$

Such a one-to-one relation between node sets $\{s_1, \ldots, s_m\}$ and $\{g_1, \ldots, g_m\}$ with an undirected edge between any two related nodes is called a *matching*. The cost of a matching are defined as the summed weights of its undirected edges. We observe that the costs of the constructed matching coincide with the cost $c$ of the considered local cycle. Conversely, having a matching with costs $c$ between node sets $\{s_1, \ldots, s_m\}$ and $\{g_1, \ldots, g_m\}$ we may always arrange matches pairs in an ordering as above, thus we obtain a local cycle with costs $c$ through node set $\{s_1, \ldots, s_m\}$. Now let us consider an arbitrary cycle cover with costs $c$ in overlap graph. Its cycles lead to a collection of (local) matchings that together form a matching with costs $c$, called a *perfect matching* ("perfect" since all nodes participate in the matching).

In this chapter we formalize the problem of finding a perfect matching with maximum weight higher or equal of a fixed threshold $t$ of a complete bipartite graph. We will present a proof of the existence of such perfect matching and we will extract a program from it. The proof-strategy we follow is simply to enumerate all the possible solutions and select the one that satisfy our constraints. This clearly generate an exponential extracted program. In our experiments we show how, applying the pruning method on special instantiations of this

problem where some additional knowledge on the input graph is assumed (the *Monge inequality*) then it is possible to extract a program extremely simplifier than the one that enumerate all the possible solutions.

## 4.2 Bounded Perfect Matching of a Complete Bipartite Graph

### 4.2.1 Basic Definitions

**Definition 4.2.1** (Weighted Bipartite Graph). *The weighted graph* $\mathcal{G} = (V \subseteq \mathbb{N},\ E \subseteq \mathbb{N}\times\mathbb{N}\times\mathbb{N})$ *is bipartite, if there exists* $V_1$ *and* $V_2$ *such that:* $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$ *and* $\forall e \in E.\, \pi_0 e \in V_1 \wedge \pi_1 e \in V_2$.

Here $\pi_{i\in\{0,1,2\}}(n_1, n_2, n_3) = n_i$, and the natural $n_3$ is the *weight* of the edge $(n_1, n_2)$. If the graph $\mathcal{G} = (V, E)$ is bipartite then we write it as $\mathcal{G} = (V_1, V_2, E)$ for two opportune sets of vertices $V_1$ and $V_2$.

**Definition 4.2.2** (Complete Weighted Bipartite Graph). *Let* $\mathcal{G} = (V_1, V_2, E)$ *be a weighted bipartite graph.* $\mathcal{G}$ *is complete if* $\forall u \in V_1\, v \in V_2\, \exists e \in E.\, \pi_0 e = u \wedge \pi_1 e = v$.

**Definition 4.2.3** (Matching). *Given the weighted bipartite graph* $\mathcal{G} = (V_1, V_2, E)$ *a matching* $M$ *of* $\mathcal{G}$ *is a subset of* $V_1 \times V_2$ *with the following two properties: for all* $u \in V_1$, $v \in V_2$ *if* $(u, v) \in M$, *then*

1. $\forall u' \in V_1 . u \neq u' \rightarrow (u', v) \notin M$
2. $\forall v' \in V_2 . v \neq v' \rightarrow (u, v') \notin M$

**Definition 4.2.4** (Weight of a Bipartite Graph). *Given a matching* $M$ *of the weighted bipartite graph* $\mathcal{G} = (V_1, V_2, E)$ *the weight of* $\mathcal{M}$, $\mathsf{SUM}(M, E)$, *is defined as follows:*

$$\begin{aligned} \mathsf{SUM}(\{\}, E) &= 0 \\ \mathsf{SUM}(M \ni e,\ E) &= v + \mathsf{SUM}(M\backslash\{e\},\ E) \text{ with } (\pi_0 e, \pi_1 e, v) \in E \end{aligned}$$

In the following we will indicate the cardinality of a set $U$ by $|U|$.

**Definition 4.2.5** (Perfect Matching). *Given a complete weighted bipartite graph* $\mathcal{G} = (V_1, V_2, E)$, *with* $|V_1| = |V_2| = n$, *we say that a matching* $M$ *of* $\mathcal{G}$ *is perfect if* $|M| = n$.

**Definition 4.2.6** (Maximum Perfect Matching Problem). *Given the complete weighted bipartite graph* $\mathcal{G} = (V_1, V_2, E)$ *with* $|V_1| = |V_2|$, *find a matching* $M$ *such that* $\mathsf{SUM}(M', E) \leq \mathsf{SUM}(M, E)$ *for any perfect matching* $M'$ *of* $\mathcal{G}$.

**Definition 4.2.7** (Bounded Perfect Matching Problem). *Given the complete weighted bipartite graph* $\mathcal{G} = (V_1, V_2, E)$ *with* $|V_1| = |V_2|$, *and* $T$ *a natural number, find a matching* $M$ *such that* $T \leq \mathsf{SUM}(M, E)$.

### 4.2.2 Algorithms, Data Structures and Automatic Program Synthesis

The sets $V_1$ and $V_2$ are implemented as lists of naturals without duplications. We indicate the length of the list $V$ by $|V|$. We indicate by $\mathsf{tail}(V)$ the operation that return the tail of the non empty list $V$. The set $E$ of weights is implemented by a list of triple of naturals $(i, j, v_{i,j})$, with $i \in V_1$, $j \in V_2$ and $v_{i,j}$ weight of the edge $(i, j)$. Given $i \in V_1$, $j \in V_2$, the weight of the arc $(i, j)$ is indicated by $E[i, j]$. A perfect matching $M$ of $V_1$ and $V_2$ is implemented by a list of naturals $M$ with the following two properties: *i*) for all $j$, if $M[j] = k$ then $(V_1[j], k)$, with $V_2[m] = k$ for some $m$, belong to the perfect matching and *ii*) for all $j \neq k$, $M[j] \neq M[k]$. By *i*) and *ii*) it follows that $M$ is a *permutation of* $V_2$. Under these assumptions the function $\mathsf{SUM} : \mathbf{N} \to \mathbf{N} \to (\mathbf{N} \times \mathbf{N} \times \mathbf{N}) \to \mathbf{N}$ (that takes in input the vector of nodes $V_1$, the matching vector $M$, the the matrix of weights $E$ and returns the weight of $M$) is defined as follow:

$$\mathsf{SUM}([], [], E) = 0$$
$$\mathsf{SUM}(v :: V_1, m :: M, E) = E[v, m] + \mathsf{SUM}(V_1, M, E)$$

Given a complete weighted bipartite graph $\mathcal{G} = (V_1, V_2, E)$, with $|V_1| = |V_2|$, $M$ is a complete matching of $\mathcal{G}$ if and only if $\mathsf{MATCH}(M, V_2)$, with the predicate $\mathsf{MATCH}$ defined as follow:

$$(M_1)\ \frac{}{\mathsf{MATCH}((:), (:))} \qquad (M_2)\ \frac{\mathsf{MATCH}(M, V_2 \backslash \{n\})}{\mathsf{MATCH}(m :: M, V_2)}\ V_2[n] = m$$

**Proposition 4.2.1.** $\forall l.\mathit{MATCH}(l, l)$

*Proof.* Case $l = (:)$, by $(M_1)$. Case $l = (a :: l')$, We have to prove $\mathsf{MATCH}(l', l')$ that follow by the induction hypothesis. □

Now we supply a constructive proof of the existence (or not) of a perfect matching (with weight higher or equal than a fixed threshold) of a complete bipartite graph. The used strategy is to *enumerate* all the possibilities, till the desired solution is found. Obviously this searching method is particularly inefficient, and it require an exponential number of steps when executed on a specific input graph.

In the rest of the chapter we will use the following conventions:

$V_1, V_2, E, T \longrightarrow M$ for "$M$ is a perfect matching between $V_1$ and $V_2$ such that $T \leq \mathsf{SUM}(V_1, M, E)$", that is $\mathsf{MATCH}(M, V_2) \wedge T \leq \mathsf{SUM}(M, V_1, E)$

$V_1, V_2, E, T \longrightarrow_n M$ for $V_1 \neq (:)$, $V_1, V_2, E, T \longrightarrow M$ and $|V_2| - n \leq \natural(M[0], V_2)$

and we wrote $V \backslash \{m\}$ to indicate the list $V$ from which is dropped the node in position $m$, with $m : 0, \ldots, |V| - 1$ and $\natural(n, V_2) = m$ for $V_2[m] = n$.

**Theorem 4.2.2.**

$$\begin{aligned}\forall V_1 V_2 E, T. \quad & (|V_1| = |V_2| \wedge 0 \leq T) \to \exists p. \\ & p \to (\exists M. V_1, V_2, E, T \longrightarrow M) \wedge \\ & (p \to \bot) \to (\exists M. V_1, V_2, E, T \longrightarrow M) \to \bot\end{aligned}$$

*Proof.* By induction on $V_1$. Case $V_1 = (:)$: Assume $V_2$,$E$, $T$ and $\mathsf{ip}{:}(|(:)| = |V_2|) \wedge 0 \leq T)$. By $\mathsf{ip}$ it follows $V_2 = (:)$. The current thesis became:

$$\begin{aligned}\exists p. \quad & p \to (\exists M. (:), (:), E, T \longrightarrow M) \wedge \\ & (p \to \bot) \to (\exists M. (:), (:), E, T \longrightarrow M) \to \bot\end{aligned} \tag{4.1}$$

Two cases are possible: Case $0 < T$: then no perfect matching does exist. So we introduce $\mathsf{ff}$ for $p$. The positive part of (4.1) is proved by ($\mathsf{Efq}$). To prove the negative part of (4.1) let's assume $\top$ and $\mathsf{ip}{:}\exists M. (:), (:), E, T \longrightarrow M$. By $\mathsf{ip}$ does exists $\overline{M}$ such that $\mathsf{MATCH}(\overline{M}, (:))$ and $T \leq \mathsf{SUM}(\overline{M}, (:), E)$. But if $\mathsf{MATCH}(\overline{M}, (:))$ by $(M_1)$ we have $\overline{M} = (:)$, and so $\mathsf{SUM}((:), (:), E) = 0$ that generate a contradiction with the hypothesis $0 < T \leq \mathsf{SUM}((:), (:), E)$. Case $0 = T$: Introduce $\mathsf{tt}$ for $p$. The positive part of (4.1) is proved introducing $(:)$ for $M$, and the negative part of (4.1) is proved by ($\mathsf{Efq}$). Case $V_1 = (a :: l)$ : Assume

$$\begin{aligned}\forall V_2 E, T. \quad & (|l| = |V_2| \wedge 0 \leq T) \to \exists p. \\ & p \to (\exists M. l, V_2, E, T \longrightarrow M) \wedge \\ & (p \to \bot) \to (\exists M. l, V_2, E, T \longrightarrow M) \to \bot\end{aligned} \tag{4.2}$$

$V_2$, $E$, $T$ and $\mathsf{ip}{:}(|(a :: l)| = |V_2| \wedge 0 \leq T)$. Given the natural $a$, we prove

$$\begin{aligned}\exists p. \quad & p \to (\exists M. (a :: l), V_2, E, T \longrightarrow M) \wedge \\ & (p \to \bot) \to (\exists M. (a :: l), V_2, E, T \longrightarrow M) \to \bot\end{aligned} \tag{4.3}$$

In order to prove (4.3) we prove the following assertion:

$$\begin{aligned}\forall n. \exists p. \quad & p \to (\exists M. (a :: l), V_2, E, T \longrightarrow_n M) \wedge \\ & (p \to \bot) \to (\exists M. (a :: l), V_2, E, T \longrightarrow_n M) \to \bot\end{aligned} \tag{4.4}$$

Obviously (4.3) is obtained instantiating (4.4) on $|V_2|$. To prove (4.4) we proceed by induction on $n$. Case $n = 0$: We shall look for a matching $M$ such that $\natural(M[0], V_2) \geq |V_2|$, but from this follow a contradiction. So we introduce $\mathsf{ff}$ for $p$. The positive part of (4.4) is proved by ($\mathsf{Efq}$). For the negative part, assume $\top$, and $\mathsf{ip}'$: $\exists M. (a :: l), V_2, E, T \longrightarrow_{|V_2|} M$. From $\mathsf{ip}'$ it follows that there exists $\overline{M}$ such that $(a :: l), V_2, E, T \longrightarrow \overline{M}$ and $|V_2| \leq \natural(\overline{M}[0], V_2)$. But $\overline{M}[0]$ is an element of $V_2$, that is $\natural(\overline{M}[0], V_2) \leq |V_2| - 1$, from which it follow a

contradiction. Now let's assume the nested inductive hypothesis

$$\begin{aligned}\exists p.\quad & p \to (\exists M.(a :: l), V_2, E, T \longrightarrow_n M) \wedge \\ & (p \to \bot) \to (\exists M.(a :: l), V_2, E, T \longrightarrow_n M) \to \bot\end{aligned} \tag{4.5}$$

a natural $n$, and we prove

$$\begin{aligned}\exists p.\quad & p \to (\exists M.(a :: l), V_2, E, T \longrightarrow_{n+1} M) \wedge \\ & (p \to \bot) \to (\exists M.(a :: l), V_2, E, T \longrightarrow_{n+1} M) \to \bot\end{aligned} \tag{4.6}$$

There are two cases, $\mathsf{Case}\, E[a, V_2[|V_2| - (n+1)]] < T$: We instantiate (4.2) on $V_2 \backslash \{|V_2| - (n+1)\}$, $E$ and $(T - E[a, V_2[|V_2| - (n+1)]])$. This instantiation produce the following hypothesis:

$$\begin{aligned}&(|l| = |V_2 \backslash \{|V_2| - (n+1)\}| \wedge 0 \leq (T - E[a, V_2[|V_2| - (n+1)]])) \to \exists p. \\ &p \to (\exists M.l, V_2 \backslash \{|V_2| - (n+1)\}, E, (T - E[a, V_2[|V_2| - (n+1)]]) \longrightarrow M) \wedge \\ &(p \to \bot) \to \\ &\quad (\exists M.l, V_2 \backslash \{|V_2| - (n+1)\}, E, (T - E[a, V_2[|V_2| - (n+1)]]) \longrightarrow M) \to \bot\end{aligned} \tag{4.7}$$

By $\mathsf{ip}$, $|(a :: l)| = |V_2|$ thus $|l| = |V_2 \backslash \{|V_2| - (n+1)\}|$, moreover by $E[a, V_2[|V_2| - (n+1)]] < T$ it follows that $0 \leq (T - E[a, V_2[|V_2| - (n+1)]])$. Instantiating (4.7) on these two facts, we know a boolean $\overline{p}$ such that

$$\begin{aligned}&\overline{p} \to (\exists M.l, V_2 \backslash \{|V_2| - (n+1)\}, E, (T - E[a, V_2[|V_2| - (n+1)]]) \longrightarrow M) \wedge \\ &(\overline{p} \to \bot) \to \\ &\quad (\exists M.l, V_2 \backslash \{|V_2| - (n+1)\}, E, (T - E[a, V_2[|V_2| - (n+1)]]) \longrightarrow M) \to \bot\end{aligned} \tag{4.8}$$

Two cases are possible, $\mathsf{Case}\, \overline{p}$: We introduce $\mathsf{tt}$ for $p$ in the goal formula (4.6) obtaining the new goal:

$$\begin{aligned}&(\mathsf{tt} \to \exists M.(a :: l), V_2, E, T \longrightarrow_{n+1} M) \wedge \\ &\bot \to ((\exists M.(a :: l), V_2, E, T \longrightarrow_{n+1} M) \to \bot)\end{aligned} \tag{4.9}$$

To prove the positive part of (4.9): assume $\mathsf{tt}$ and instantiate the left of (4.8) on $\overline{p}$, from which it follow that there exists $\overline{M}$ such that:

$$l, V_2 \backslash \{|V_2| - (n+1)\}, E, (T - E[a, V_2[|V_2| - (n+1)]]) \longrightarrow \overline{M} \tag{4.10}$$

So we introduce $(V_2[|V_2| - (n+1)] :: \overline{M})$ for $M$. We have to prove:

- $\mathsf{MATCH}((V_2[|V_2| - (n+1)] :: \overline{M}), V_2)$: by $(M_2)$ this correspond to prove $\mathsf{MATCH}(\overline{M}, V_2 \backslash \{|V_2| - (n+1)\})$, that hold by (4.10).
- $T \leq \mathsf{SUM}(a :: l, (V_2[|V_2| - (n+1)] :: \overline{M}), E)$: This correspond to prove $T - E[a, V_2[|V_2| - (n+1)]] \leq \mathsf{SUM}(l, \overline{M}, E)$, that follow by (4.10).
- $|V_2| - (n+1) \leq \natural((V_2[|V_2| - (n+1)] :: \overline{M})[0], V_2)$: By definition $(V_2[|V_2| -$

$(n+1)] :: \overline{M})[0] = V_2[|V_2| - (n+1)]$ and then $\natural(V_2, V_2[|V_2| - (n+1)]) = |V_2| - (n+1)$.

To prove the negative part of (4.9): by (Efq). Case $\overline{p} \to \bot$: by (4.5) there exists $\overline{\overline{p}}$ such that:

$$\begin{array}{l} \overline{\overline{p}} \to (\exists M.(a :: l), V_2, E, T \longrightarrow_n M) \wedge \\ (\overline{\overline{p}} \to \bot) \to (\exists M.(a :: l), V_2, E, T \longrightarrow_n M) \to \bot \end{array} \tag{4.11}$$

We introduce $\overline{\overline{p}}$ for $p$ in (4.6) obtaining the new goal:

$$\begin{array}{l} \overline{\overline{p}} \to (\exists M.(a :: l), V_2, E, T \longrightarrow_{n+1} M) \wedge \\ (\overline{\overline{p}} \to \bot) \to (\exists M.(a :: l), V_2, E, T \longrightarrow_{n+1} M) \to \bot \end{array} . \tag{4.12}$$

To prove the positive part of (4.12): assume $\overline{\overline{p}}$ and instantiate the positive part of (4.11) on $\overline{\overline{p}}$. It follows that there exists $\overline{M}$ perfect matching between $(a :: l)$ and $V_2$ such that $|V_2| - n \leq \natural(\overline{M}[0], V_2)$, and thus, $|V_2| - (n+1) \leq \natural(\overline{M}[0], V_2)$. To prove the negative part of (4.12): Assume $\overline{\overline{p}} \to \bot$. Now, considering that:

- Instantiating the negative part of (4.8) on $(\overline{p} \to \bot)$ there not exists any $M$ such that $l, V_2 \backslash \{|V_2| - (n+1)\}, E, (T - E[a, V_2[|V_2| - (n+1)]]) \longrightarrow M$. Thus, for each matching $M$, naming $\delta_M = \mathsf{SUM}(l, M, E)$, we have $\delta_M < T - E[a, V_2[|V_2| - (n+1)]]$ and thus $\delta_M + E[a, V_2[|V_2| - (n+1)]] < T$, i.e. there exists no matching $M$ between $(a :: l)$ and $V_2$ such that $M[0] = V_2[|V_2| - (n+1)]$.

- Instantiating the negative part of (4.11) on $\overline{\overline{p}} \to \bot$ there not exists any $M$ such that: $(a :: l), V_2, E, T \longrightarrow_n M$

we conclude that there exists no matching $M$ such that: $(a :: l), V_2, E, T \longrightarrow_{n+1} M$. Case $E[a, V_2[|V_2| - (n+1)]] \geq T$: The value of the matching built so far is higher than $T$, so we can stop the search. Instantiate (4.2) on $V_2 \backslash \{|V_2| - (n+1)\}$, $E$ and 0. Thus there exists $\overline{p}$ such that

$$\begin{array}{l} \overline{p} \to (\exists M.l, V_2 \backslash \{|V_2| - (n+1)\}, E, 0 \longrightarrow M) \wedge \\ (\overline{p} \to \bot) \to (\exists M.l, V_2 \backslash \{|V_2| - (n+1)\}, E, 0 \longrightarrow M) \to \bot \end{array} \tag{4.13}$$

Each matching between two set has a value greater or equal than zero (except the case in which we consider arcs with negative weight). Thus $\overline{p}$ *has to be* true. We state this fact asserting the validity of $\overline{p}$. So we have two new goals: $\overline{p}$ and $\overline{p} \to$(4.6). To prove $\overline{p}$: we assert that the existence a matching between $l$ and $V_2 \backslash \{|V_2| - (n+1)\}$ with a value higher or equal than 0. We create two new subgoals:

$$\exists M.l, V_2 \backslash \{|V_2| - (n+1)\}, E, 0 \longrightarrow M \tag{4.14}$$

$$(\exists M.l, V_2\backslash\{|V_2| - (n+1)\}, E, 0 \longrightarrow M) \to \overline{p}. \tag{4.15}$$

To prove (4.14): by definition the returning matching, if it exists, is a list of naturals, permutation of $V_2\backslash\{|V_2| - (n+1)\}$. So we can return the *identity* permutation, that is we introduce $V_2\backslash\{|V_2| - (n+1)\}$ for $M$. We have to prove:

- $\mathsf{MATCH}(V_2\backslash\{|V_2| - (n+1)\}, V_2\backslash\{|V_2| - (n+1)\})$: By Prop. 4.2.1.
- $0 \leq \mathsf{SUM}(l, V_2\backslash\{|V_2| - (n+1)\}, E)$: By definition of $\mathsf{SUM}$.

To prove (4.15): Assume $\mathsf{ip}' : \exists M.l, V_2\backslash\{|V_2| - (n+1)\}, E, 0 \longrightarrow M$. The hypothesis (4.13) is a conjunction, so both the branches have to be true. In particular, by $\mathsf{ip}$, $(\exists M.l, V_2\backslash\{|V_2| - (n+1)\}, E, 0 \longrightarrow M) \to \bot$ is false, thus $\overline{p} \to \bot$ has to be false and $\overline{p}$ true. To prove $\overline{p} \to$ (4.6). Assume $\overline{p}$. We introduce $\mathsf{tt}$ for $p$ in (4.6) obtaining

$$\begin{array}{l} \mathsf{tt} \to (\exists M.(a :: l), V_2, E, T \longrightarrow_{n+1} M) \wedge \\ \bot \to (\exists M.(a :: l), V_2, E, T \longrightarrow_{n+1} M) \to \bot \end{array} \tag{4.16}$$

To prove the positive part of (4.16): assume $\mathsf{tt}$. Instantiate the left of (4.13) on $\overline{p}$, so we know $\overline{M}$ such that:

$$l, V_2\backslash\{|V_2| - (n+1)\}, E, 0 \longrightarrow \overline{M} \tag{4.17}$$

We introduce $(V_2[|V_2| - (n+1)] :: \overline{M})$ for $M$. We have to prove:

- $\mathsf{MATCH}(V_2[|V_2| - (n+1)] :: \overline{M}, V_2)$: By $(M_2)$ it corresponds to prove $\mathsf{MATCH}(\overline{M}, V_2\backslash\{|V_2| - (n+1)\})$ that follow by (4.17).
- $T \leq \mathsf{SUM}((a :: l), V_2[|V_2| - (n+1)] :: \overline{M}, E)$: It is equivalent to prove $T \leq E[a, V_2[|V_2| - (n+1)]] + \mathsf{SUM}(l, \overline{M}, E)$. This fact follows from $\mathsf{SUM}(l, \overline{M}, E) \geq 0$, by (4.17), and by the hypothesis $E[a, V_2[|V_2| - (n+1)]] \geq T$.
- $|V_2| - (n+1) \leq \natural((V_2[|V_2| - (n+1)] :: \overline{M})[0], V_2)$: already proved as a valid inequality.

To prove the negative part of (4.16): by $(\mathsf{Efq})$. □

The computational content of the Theorem4.2.2 is showed in Table 4.1 (the algorithm is written by metarules)

### 4.2.3 Problem Specialization: The Monge Inequality

In this subsection we present an algorithm to solve the bounded perfect matching problem in presence of additional knowledge on the input parameters. The

$$\frac{}{(:), V_2, E, T \longrightarrow (\mathsf{ff}, (:))}\ 0 < T \qquad \frac{}{(:), V_2, E, 0 \longrightarrow (\mathsf{tt}, (:))}$$

$$\frac{(a :: l), V_2, E, T \longrightarrow_{|V_2|} (p, M)}{(a :: l), V_2, E, T \longrightarrow (p, M)} \qquad \frac{}{V_1, V_2, E, T \longrightarrow_0 (\mathsf{ff}, (:))}$$

$$\frac{\mathsf{tail}(V_1), V_2 \backslash \{|V_2| - (n+1)\}, E, T' \longrightarrow (\mathsf{tt}, M)}{V_1, V_2, E, T \longmapsto_{n+1} (\mathsf{tt}, (V_2[|V_2| - (n+1)] :: M))}\ \mathsf{C}_1$$

$$\frac{\mathsf{tail}(V_1), V_2 \backslash \{|V_2| - (n+1)\}, E, T' \longrightarrow (\mathsf{ff}, \_) \qquad V_1, V_2, E, T \longrightarrow_n (p, M)}{V_1, V_2, E, T \longmapsto_{n+1} (p, M)}\ \mathsf{C}_1$$

$$\frac{\mathsf{tail}(V_1), V_2 \backslash \{|V_2| - (n+1)\}, E, 0 \longrightarrow (p, M)}{V_1, V_2, E, T \longmapsto_{n+1} (\mathsf{tt}, (V_2[|V_2| - (n+1)] :: M))}\ \neg\mathsf{C}_1$$

with
$T' := T - E[V_1[0], V_2[|V_2| - (n+1)]]$,
$\mathsf{C}_1 := E[V_1[0], V_2[|V_2| - (n+1)]] < T$,
$\neg\mathsf{C}_1 := E[V_1[0], V_2[|V_2| - (n+1)]] \not< T$

Table 4.1: Algorithm to compute the Maximum Perfect Matching of the Bipartite Graph $\mathcal{G} = (V_1, V_2, E)$ ($V_1$ and $V_2$ of same cardinality)

solution we present here is not synthesized from a proof, anyway the correctness of the method is proved formally. The basic idea is that if the input bipartite graph $\mathcal{V}$ satisfy a certain property, the *Monge inequality*, then we can compute the weight of the maximum perfect matching of $\mathcal{V}$ using a particularly fast algorithm, called the *Greedy algorithm* [20]. Once we have $v_{max}$, weight of the maximum perfect machting of $\mathcal{V}$, and $t$ natural threshold, then the bounded perfect matching problem can be solved by just a comparison between $v_{max}$ and $t$.

**Definition 4.2.8.** *Let $\mathcal{G} = (V_1, V_2, E)$ a complete weighted bipartite graph. Now let $u, u' \in V_1$ and $v, v' \in V_2$ and assume without loss of generality that $\mathsf{max}\{E[u, v'], E[u', v], E[u', v']\} \leq E[u, v]$. If*

$$E[u', v] + E[u, v'] \leq E[u, v] + E[u', v']$$

*then the four nodes $u, v, u', v'$ are said to satisfy the Monge inequality.*

A complete weighted bipartite graph is said to satisfy the Monge inequalities if the Monge inequality is satisfied for any two arbitrary nodes from $V_1$ together with any two arbitrary nodes from $V_2$.

If a complete weighted bipartite graph $\mathcal{G}$ satisfy the Monge inequalities then the *Greedy Assignment* algorithm (Figure 4.1) applied to $\mathcal{G}$ return a maximal matching for $\mathcal{G}$ (theorem 4.2.3). This is not true in general [For example consider $\mathcal{G} = (\{0,1\}, \{0,1\}, \{(0,0,9), (0,1,10), (1,0,1), (1,1,7)\})$].

The greedy assignment algorithm runs in $\mathcal{O}(n^2 log(n))$, with $V_i = n$, and is one the known fastest algorithm to compute the maximal perfect matching of $\mathcal{G}$ for $\mathcal{G}$ complete weighted bipartite graph that satisfy the Monge inequality.

In Figure 4.1 we used the following notation:

- $l_1, l_2 \longmapsto_{\mathsf{m}} l_3$, for " $l_3$ is the *ordered* merge of the two lists of weighted nodes $l_1$ and $l_2$"
- $E \longmapsto_{\mathsf{ms}} E'$, for "$E'$ is obtained ordering the list of weighted nodes $E$ by the Merge Sort algorithm"
- $V_1, V_2, E \longmapsto_{\mathsf{gr}} M$, for "$M$ is the perfect matching between $V_1$ and $V_2$ obtained by the Greedy assignment ".
- $E\{\rightarrow v\}$ for the list $E$ where are dropped all the arcs $(u, v, l_{u,v})$, for each $u \in V_1$
- $E\{u \leftarrow\}$ for the list $E$ where are dropped all the arcs $(u, v, l_{u,v})$, for each $v \in V_2$
- $E_{[i,\ldots,j]} = (E[i], \ldots, E[j])$, $E_{[|E|,\ldots,|E|-1]} = (:)$.

**Theorem 4.2.3.** *Given a complete weighted bipartite graph $\mathcal{G} = (V_1, V_2, E)$, if $V_1, V_2, E \longmapsto_{gr} M$ then $M$ is a maximum perfect matching of $\mathcal{G}$.*

*Proof.* By contradiction. Assume $\overline{M}$ is a perfect matching between $V_1$ and $V_2$ with respect to the set of edges $E$ such that $\mathsf{SUM}(M) < \mathsf{SUM}(\overline{M})$. Then assume $e = (u, v) \in M$ first edge found by the greedy algorithm that *does not* belong to $\overline{M}$. Follow that the two edges $a = (u, v')$, and $b = (u', v)$, has to belong to $\overline{M}$, for some $u' \in V_1$, $v \neq u'$, $u' \in V_2$, $v \neq v'$,. The edge $e$ was chosen by the greedy algorithm among also all the edges incident in $u$ and $v$, so $E[e] \geq E[a]$ and $E[e] \geq E[b]$. Now the cases are possible:

$f = (u', v') \in M$ , Obviously $f \notin \overline{M}$. Being $e$ the first edge chosen by the greedy algorithm not in $\overline{M}$ then $E[e] \geq E[f]$ and by the Monge inequality,

$$\frac{}{l, (:) \longmapsto_{\mathsf{m}} l} \qquad \frac{}{(:), l \longmapsto_{\mathsf{m}} l}$$

$$\frac{l_1, ((u', v', l'_{u,v}) :: l_2) \longmapsto_{\mathsf{m}} l_3}{((u, v, l_{u,v}) :: l_1), ((u', v', l'_{u,v}) :: l_2) \longmapsto_{\mathsf{m}} ((u, v, l_{u,v}) :: l_3)}\ l_{u,v} \leq l'_{u,v}$$

$$\frac{((u, v, l_{u,v}) :: l_1), l_2 \longmapsto_{\mathsf{m}} l_3}{((u, v, l_{u,v}) :: l_1), ((u', v', l'_{u,v}) :: l_2) \longmapsto_{\mathsf{m}} (((u', v', l'_{u,v}) :: l_3)}\ l_{u,v} > l'_{u,v}$$

$$\frac{E_{[0,\ldots,\lfloor \frac{|E|}{2} \rfloor - 1]} \longmapsto_{\mathsf{ms}} l_1 \quad E_{[\lfloor \frac{|E|}{2} \rfloor, \ldots, |E|-1]} \longmapsto_{\mathsf{ms}} l_2 \quad l_1, l_2 \longmapsto_{\mathsf{m}} l_3}{E \longmapsto_{\mathsf{ms}} l_3}\ |E| > 0,$$

$$\frac{}{(:) \longmapsto_{\mathsf{gr}'} (:)} \qquad \frac{l\{\rightarrow v\}\{u \leftarrow\} \longmapsto_{\mathsf{gr}'} M}{((u, v, l_{u,v}) :: l) \longmapsto_{\mathsf{gr}'} ((u, v, l_{u,v}) :: M)}$$

$$\frac{}{E \longmapsto_{\mathsf{ms}} (:)}\ |E| = 0 \qquad \frac{E \longmapsto_{\mathsf{ms}} l \qquad l \longmapsto_{\mathsf{gr}'} M}{V_1, V_2, E \longmapsto_{\mathsf{gr}} M}$$

$$\frac{V_1, V_2, E \longmapsto_{\mathsf{gr}} M}{V_1, V_2, E, T \longmapsto_{\mathsf{bgr}} (\mathsf{tt}, M)}\ \mathsf{C} \qquad \frac{V_1, V_2, E \longmapsto_{\mathsf{gr}} M}{V_1, V_2, E, T \longmapsto_{\mathsf{bgr}} (\mathsf{ff}, (:))}\ \neg\mathsf{C}$$

with $\mathsf{C} \equiv T \leq \mathsf{SUM}(M, E)$

Figure 4.1: Greedy Assignment to find the Bounded Perfect Matching of a bipartite Graph $\mathcal{G} = (V_1, V_2, E)$ that satisfy the Monge inequality.

$E[a] + E[b] \leq E[e] + E[f]$. We define the new perfect matching $\overline{M}'$ by a "local" modification of $\overline{M}$ as follow:

$$\overline{M}' = (\overline{M} \backslash \{a, b\}) \cup \{e, f\}$$

Being $e, f$ in $M$ we have $|M \backslash \overline{M}'| = |M \backslash \overline{M}| - 2$, that is $|M \backslash \overline{M}'| < |M \backslash \overline{M}|$. Moreover by the inequality $E[a] + E[b] \leq E[e] + E[f]$ we have $\mathsf{SUM}(\overline{M}) \leq \mathsf{SUM}(\overline{M}')$.

$f = (u', v') \notin M$ , Let's assume $h = (u', v'') \in M$ and $g = (v', u'') \in M$ with $v' \neq v''$ and $u' \neq u'$. Obviously $h, g \notin \overline{M}$ else would be violated the property to be a matching for $\overline{M}$, and being $e$ the first edge in $M \backslash \overline{M}$

chose by the greedy algorithm we have $E[h] \leq E[e]$ and $E[g] \leq E[e]$. If $h, g$ are picked up by the greedy algorithm then $E[f] \leq E[h]$, $E[f] \leq E[g]$ and by transitivity $E[f] \leq E[e]$. Thus by the monge inequality $E[a] + E[b] \leq E[e] + E[f]$. As in the above case we define the new perfect matching $\overline{M}'$ by a "local" modification of $\overline{M}$ as follow:

$$\overline{M}' = (\overline{M} \backslash \{a, b\}) \cup \{e, f\}$$

Being $e$ in $M$ we have $|M \backslash \overline{M}'| = |M \backslash \overline{M}| - 1$, that is $|M \backslash \overline{M}'| < |M \backslash \overline{M}|$. Moreover by the inequality $E[a] + E[b] \leq E[e] + E[f]$, $\mathsf{SUM}(\overline{M}) \leq \mathsf{SUM}(\overline{M}')$.

Now if $M = \overline{M}'$ stop, else we set $\overline{M} \leftarrow \overline{M}'$, pick a new $e$ first edge in $M \backslash \overline{M}$ and repeat the procedure above. This algorithm produce a list of perfect matchings $\overline{M}_1, \overline{M}_2, \ldots, \overline{M}_{d \leq n}$ (in the beginning we have $\overline{M} = \overline{M}_1$) such that

$$0 = |M \backslash \overline{M}_d| < \ldots < |M \backslash \overline{M}_2| < |M \backslash \overline{M}_1|$$

(from which follow $M = \overline{M}_d$) and

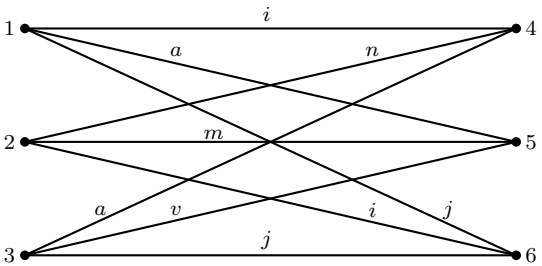
$$\mathsf{SUM}(M) < \mathsf{SUM}(\overline{M}_1) \leq \ldots \leq \overline{M}_d = \mathsf{SUM}(M)$$

□

## 4.3 Pruning at Work

In this section we show the results we obtained applying the pruning protocol to the proof of Theorem 4.2.2.

**Proof Specialization** We specialized the proof on the following complete bipartite graph $\mathcal{V}$:



That is, $\mathcal{V} = (\{1,2,3,4,5,6\}, \{(1,4,i),(1,5,a),(1,6,j),(2,4,n),(2,5,m),(2,6,i),(3,4,a),(3,5,v),(3,6,j)\})$.

**Pruning** We applied a first time pruning on the specialized proof in order to manipulate a shorter proof during the pruning protocol. The extracted code is here listed:

```
[if (0<t--i--m--j)
 [if (0<t--i--i--v)
  [if (0<t--a--n--j)
   [if (0<t--a--i--a)
    [if (0<t--j--n--v)
     [if (0<t--j--m--a)
      (False@(Nil nat))
      (True@
      6::
      [if (0<t--j--n--v)
        [if (0<t--j--m--a)(Nil nat)
                                (5::[if(0<t--j--m--a)(Nil nat)(4:)])]
        (4::[if(0<t--j--n--v)(Nil nat)(5:)])])]
     (True@
     6::
     [if(0<t--j--n--v)
      [if(0<t--j--m--a)(Nil nat)(5::[if(0<t--j--m--a)(Nil nat)(4:)])]
      (4::[if (0<t--j--n--v) (Nil nat) (5:)])])]
    (True@
    5::
    [if (0<t--a--n--j)
      [if(0<t--a--i--a)(Nil nat)(6::[if(0<t--a--i--a)(Nil nat)(4:)])]
      (4::[if (0<t--a--n--j) (Nil nat) (6:)])])]
   (True@
   5::[if (0<t--a--n--j)
     [if(0<t--a--i--a)(Nil nat)(6::[if (0<t--a--i--a)(Nil nat)(4:)])]
     (4::[if (0<t--a--n--j) (Nil nat) (6:)])])]   (True@   4::
  [if (0<t--i--m--j)
    [if (0<t--i--i--v) (Nil nat)(6::[if(0<t--i--i--v)(Nil nat)(5:)])]
    (5::[if (0<t--i--m--j) (Nil nat) (6:)])])]  (True@
  4::  [if (0<t--i--m--j)
   [if(0<t--i--i--v)(Nil nat)(6::[if (0<t--i--i--v) (Nil nat) (5:)])]
    (5::[if (0<t--i--m--j) (Nil nat) (6:)])])]
```

**Permutative Conversion** At this stage we permuted the pruned proof of the previous step. As extensively explained in chapter 3.2.3 this operation is necessary if we want to perform successfully the *dependencies removal transformation* step of the pruning protocol. Moreover, as we will see, to permute a proof it has the nice side effect of eliminating part of the redundancies in the extracted code. It follow the code synthesized from permuted proof:

```
[if (0<t-i-m-j)
  [if (0<t-i-i-v)
   [if (0<t-a-n-j)
    [if (0<t-a-i-a)
     [if (0<t-j-n-v)
      [if (0<t-j-m-a)
       (False@(Nil nat))
       (True@6::[if (0<t-j-m-a) (Nil nat) (5::4:)])]
      (True@6::4::5:)]
     (True@5::[if (0<t-a-i-a) (Nil nat) (6::4:)])]
    (True@5::4::6:)]
   (True@4::[if (0<t-i-i-v) (Nil nat) (6::5:)])]
  (True@4::5::6:)]
```

**Removal Dependencies Transformation** The code extracted in the previous step still contain several redundancies, as for example the presence of several nested `if`'s statements on the same boolean condition. This kind of redundancies are even more if we assume to have some knowledge on the input weights of the complete graph $\mathcal{V}$. In this particular case study we assumed the input graph $\mathcal{V}$ to satisfy the *Monge inequality*. More precisely we assumed the following inequalities relations among the weights of the input graph:

1. $m < i < n < a < v < j$
2. $m + 1 \leq a + m$
3. $i + a \leq m + j$
4. $2i \leq j + n$
5. $a + m \leq v + n$
6. $a + i \leq j + n$
7. $i + v \leq j + m$
8. $2a \leq v + i$

At this point the *removal dependencies transformation* was applied keeping into account the additional knowledge on $\mathcal{V}$. After that, pruning was applied again. It follow the extracted program of the resulting proof:

```
(*) [if (0<t-i-m-j)
     [if (0<t-a-n-j)
       [if (0<t-j-n-v) (False@(Nil nat)) (True@6::4::5:)]
       (True@5::4::6:)]
     (True@4::5::6:)]
```

We note that this code is extremely shorter than the code we synthesized after the **proof specialization** step. Anyway, in order have a better

comprehension of the quality of our result, we instantiated the program in Figure ...(the better algorithm to compute a solution for the bounded perfect matching problem) on the input graph $\mathcal{V}$ and later on we simplified it according to the above constraints $1., \ldots, 8$. The resulting program is the following:

```
(**) [if (0<t-m-i-j) (False@(Nil nat)) (True@4::5::6:)]
```

As we can see, if the input parameter `t` is less or equal than $t_{max}$, with $t_{max}$ weight of the perfect matching of $\mathcal{V}$ with maximum weight, then both `(*)` and `(**)` returns in one step the couple `(True@4::5::6:)`. On the other hand if `t` $> t_{max}$, that is the problem does not admit a solution, `(**)` return `(False@(Nil nat))` in one step while `(*)` needs to perform, in order to return the same result, two more case distinctions. This phenomena rely essentially on the fact that no one of the eight constraints $1., \ldots, 8.$ involve the parameter `t`

## 4.4 Conclusions

What we showed in this chapter is that the pruning protocol *matters* in the automatic synthesis of correct and efficient code. Starting from a proof of an existential statement proved by an enumeration strategy (from which it was possible to synthesize an algorithm with an exponential complexity running time) we were able to produce, through the several proof refinements steps of the pruning technique, a new proof of an instance of the original problem with computational content *comparable* with the instantiation of a quadratic running time algorithm that solved the same problem.

The main limit of the present work is the restricted set of input graphs on which we could test the pruning protocol, but are working in order to extend this set of examples in order to have a cleared idea of the power of this method.

# 5 Generalizing Pruning

## 5.1 Introduction

In Chapters 3 and 4 we have introduced the Pruning technique and we have shown the power of this proof transformation on two particularly big examples: the Bin Packing Problem and the Bounded Perfect Matching Problem. We have seen that the transformations pruning induce on the extracted programs could not be performed by any other known program transformation: pruning manipulates the proofs of the programs, so it works with dependencies informations that does not occur in programs written by people. In this chapter we present an extension of the pruning technique and we will show its effectiveness on a very simple but instructive example.

## 5.2 Proof Contexts

Here we define $\lambda$-terms $C^A_B$ for natural deduction proofs of type $A$ with exactly one *hole* ($\bullet$) of type $B$.

*Definition* 5.2.1 (Proof Context).

$$\begin{aligned} C \quad := \quad & \bullet^B \mid \langle M^A, C^B \rangle^{A \wedge B} \mid \langle C^A, M^B \rangle^{A \wedge B} \mid (\pi_0 C^{A \wedge B})^A \mid (\pi_1 C^{A \wedge B})^B \mid \\ & (\lambda x^\rho C^A)^{\forall x^\rho A} \mid (C^{\forall x^\rho A} t)^{A[x/t]} \mid (M^{A \to B} C^A)^B \mid (C^{A \to B} M^A)^B \mid \\ & (\lambda u^A C^B)^{A \to B} \end{aligned}$$

By $C^A_B[M^B]$ we indicate the replacing of $\bullet^B$ in $C^A$ with $M^B$.

*Definition* 5.2.2. Let $M^A$ and $N^A$ be two proofs of the same formula $A$ with the property that there exists a proof context $C^A$ such that $M^A \equiv C^A[N^A]$ (syntactic equivalence). The set of *discharged* assumptions from $N$ to $M$, $\mathsf{DSA}(C)$, is defined as follow:

$$\mathsf{DSA}(\bullet) \quad = \quad \emptyset$$

$$\mathsf{DSA}(C) \quad = \quad \begin{cases} \mathsf{DSA}(C') \cup \{u\} & C \equiv C'[\lambda u^A \bullet] \\ \mathsf{DSA}(C') & C \equiv C'[(\pi_0 \bullet^{A \wedge B})^A], C'[(\pi_1 \bullet^{A \wedge B})^B], \\ & \quad C'[\langle M^A, \bullet^B \rangle^{A \wedge B}], C'[\langle \bullet^A, M^B \rangle^{A \wedge B}] \\ & \quad C'[(\bullet^{\forall x^\rho A} t)^{A[x/t]}], C'[(M^{A \to B} \bullet^A)^B], \\ & \quad C'[(\bullet^{A \to C} M^A)^B], C'[\lambda x^\rho \bullet], \end{cases}$$

for some opportune $C'$.

Based on the previous definitions, in Figure 5.1 we propose the *general* pruning rule. The redundancies eliminated by the simplification rule in Figure

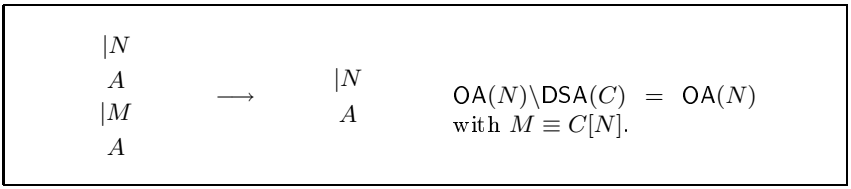



Figure 5.1: *General* pruning rule

5.1 are not so obvious to find in proof written by hand but not rely such inferences occurs in proofs generated automatically by automatic theorem provers or in proofs where part of the input parameters are specialized.

## 5.3 Properties of the General Pruning Rule

We will write $M \longrightarrow_p N$ for "$N$ is obtained from $M$ applying one of the pruning rules in Figure 3.1", $M \longrightarrow_{gp} N$ for "$N$ is obtained from $M$ applying the general pruning rule in Figure 5.1", and $\longrightarrow_p^+$ and $\longrightarrow_{gp}^+$ for the transitive closure of $\longrightarrow_p$ and $\longrightarrow_{gp}$. Given the derivations $M$ and $N$ we define $M \xrightarrow{n}_p N$, with $0 < n$, as follow:

- If $M \longrightarrow_p N$ then $M \xrightarrow{1}_p N$
- If $M \longrightarrow_p M' \longrightarrow_p^+ N$ and $M' \xrightarrow{n}_p N$ then $M \xrightarrow{n+1}_p N$

Being not unique the derivation between $M$ and $N$ (there could be many) there will be different "$n$" such that $M \xrightarrow{n}_p N$.

**Proposition 5.3.1.** *For each proof context $C_A^C$, proofs $M^A$ and $N^A$, if $M \longrightarrow_p N$ then $C[M] \longrightarrow_p C[N]$.*

The same hold for $\longrightarrow_{gp}$. Obviously if we locally simplify a proof by $\longrightarrow_p$ then the same simplification can be performed by the general pruning rule $\longrightarrow_{gp}$. This fact is stated in the following

**Theorem 5.3.2.** *For all proofs $M$ and $N$, if $M \longrightarrow_p N$ then $M \longrightarrow_{gp} N$.*

*Proof.* If $M \longrightarrow_p N$, then only three cases are possible:

1. $M \equiv C[N']$ with $N' \equiv (\mathsf{IF}\, t\, \lambda u^t M'^C\ \lambda v^{t\to\bot} N''^C)^C$, for an opportune context proof $C$, and $u^t$ does not occur in $M'$. Then $C[N'] \longrightarrow_p C[M']$ by the rule $i$), Figure 3.1. On the other hand we can write $N' \equiv C'[M']$ with $C' \equiv (\mathsf{IF}\, t\, \lambda u^t \bullet^\phi\ \lambda v^{t\to\bot} N''^\phi)^\phi$. It follow that $\mathsf{OA}(M')\backslash\mathsf{DSA}(C') = \mathsf{OA}(M')$ because $\mathsf{DSA}(C') = \{u\}$ and by hypothesis $u \notin \mathsf{OA}(M')$. So applying the simplification rule in Figure 5.1 to $N'$ we have $C[C'[M']] \longrightarrow_{gp} C[M']$

2. $M \equiv C[N']$ with $N' \equiv (\mathsf{IF}\ t\ \lambda u^t M'^C\ \lambda v^{t\to\bot} N''^C)^C$ and $v^{t\to\bot}$ does not occur in $N''$. We proceed as in the previous case (but we use the pruning rule $ii$) instead of $i$)).

3. $M \equiv C[N']$ with $N' \equiv (\exists^-_{x,A,C} M'^{\exists x A}\ \lambda x^\rho u^A N''^C)^C$ and $u^A$ does not occur in $N''$. We proceed as in point 1. (and we use the pruning rule $iii$) instead of $i$)).

□

**Theorem 5.3.3.** *For all $n$, $M$ and $N$, if $M \xrightarrow{n+1}_p N$ then $M \longrightarrow^+_{gp} N$.*

*Proof.* By induction on $n$. If $n = 0$ we have $M \longrightarrow_p N$ and thus $M \longrightarrow_{gp} N$ by Theorem 5.3.2, and $M \longrightarrow^+_{gp} N$ by definition of $\longrightarrow^+_{gp}$. Now assume that for each $M$ and $N$, if $M \xrightarrow{n+1}_p N$ then $M \longrightarrow^+_{gp} N$ and assume $M' \longrightarrow_p \overline{M} \xrightarrow{n+1}_p N'$, for some fixed $M'$, $\overline{M}$ and $N'$. Instantiating the induction hypothesis on $\overline{M}$ and $N'$, we have $\overline{M} \longrightarrow^+_{gp} N'$. By Theorem 5.3.2 if $M \longrightarrow_p \overline{M}$ then $M \longrightarrow_{gp} \overline{M}$ and hence $M \longrightarrow^+_{gp} \overline{M}$ and by transitivity $M \longrightarrow^+_{gp} N'$. □

**Corollary 5.3.4.** *For all $M$ and $N$, if $M \longrightarrow^+_p N$ then $M \longrightarrow^+_{gp} N$.*

On the other hand, it is not true that we can mimic any reduction performed by $\longrightarrow_{gp}$ with $\longrightarrow_p$ (for this reason the implication in Theorem 5.3.2 is not an equivalence). Consider for example the following proof:

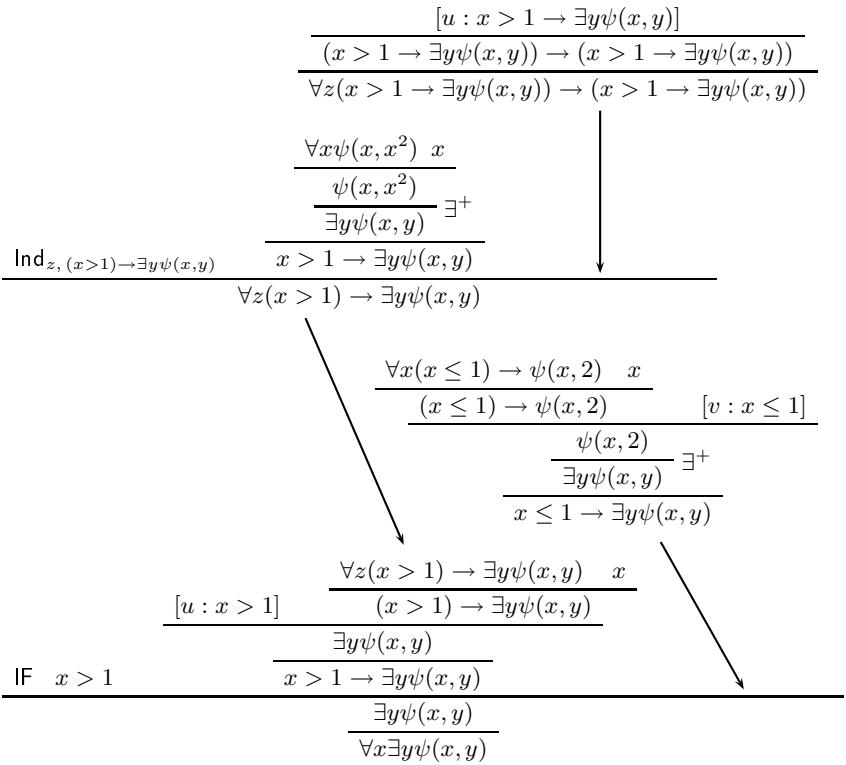
$$\dfrac{\mathsf{IF}\quad t \qquad \dfrac{\begin{array}{c}|M\\ A\end{array}}{t \to A} \qquad \dfrac{\begin{array}{c}|N'\\ A\end{array}}{(t \to \bot) \to A}}{A}$$

and assume that the assumption $u^t$ occur in $N$ but not in $M$, that $v^{t\to\bot}$ occur in $N'$ and finally that no open assumption of $M$ is discharged in $N$. Under these conditions the pruning rules (Figure 3.1) are not applicable. For the contrary using the general pruning rules (Figure 5.1) the above proof reduce to the simplest:

$$|M$$
$$A$$

## 5.4 Case Study

Consider the predicate $\psi \subseteq \mathbf{N} \times \mathbf{N}$ such that $\psi(x,y) \Leftrightarrow x^2 \leq y$. We propose the following *original* derivation of the fact that for each natural $x$ there exist a natural $y$ such that $\psi(x,y)$. Through the proof we will make use of the following axioms: $\forall x \psi(x, x^2)$ and $\forall x (x \leq 1) \to \psi(x,2)$.




The code extracted from the previous proof is the following:

$$\lambda x \ \mathsf{if}\,(x > 1)\,((\mathcal{R}^{\mathbf{N}}_{\mathbf{N}}\ x^2\ \lambda n, p.\, p)\, x)\ 2$$

Obviously, for $x > 1$, this code perform useless computation in order to compute $x^2$, but more important, no redundancies are detect in the proof by the pruning

rules (Figure 3.1). In fact, both the assumption variables $u$ and $v$ occurs respectively in the left and right branches of the case distinction.

On the other hand we see that in the base case of the induction we prove the formula $\exists y \psi(x, y)$ without using $u$ and none of the assumptions used in this subproof is later on discharged through the path to the other occurrences of $\exists y \psi(x, y)$ at the end of the case distinction. Under these conditions we can apply the general pruning rule (Figure5.1) to our proof obtaining:

$$\dfrac{\dfrac{\dfrac{\forall x \psi(x, x^2) \;\; x}{\psi(x, x^2)}}{\exists y \psi(x, y)}}{\forall x \exists y \psi(x, y)}$$

From which it is possible to extract the simplified code: $\lambda x x^2$.

# 6 String Alignment

## 6.1 Introduction

A widely studied problem in Bioinformatics is to find the *distance* between two given sequences of symbols (over an alphabet $\Sigma$). The two main techniques developed in this area to solve this problem turned out to be the *edit distance* and the *similarity* of strings [20].

*Edit distance* focus on the transformation of the first list into the second one using a restricted set of operations (insertion $I$, deletion $D$, matching $M$, and replacement $R$) Given two lists we define the *edit distance problem* the task of finding the minimum number of insertions, deletions and substitutions operations to transform the first list to the second one. Once the right set of basic operation is found, this is stored in a string called *edit transcript* (build on the alphabet $I$,$D$,$M$, and $R$) that will constitute the output of the problem (Figure 6.1, line 1).

The other way to measure the distance of lists is the so called *similarity* method. The idea is based on the concept of *string alignment*. Given two strings $l_1$ and $l_2$, *an* alignment of $l_1$ and $l_2$ is obtained inserting a new symbol "_" (named *space*) (that does not belong to $\Sigma$) into the strings $l_1$ and $l_2$ and then placing the two strings one above the other, so that every character or space in either list is opposite a unique character or space in the other list, and no space is opposite to another space (Figure 6.1, lines 2,3). We indicate by $(\delta_1, \delta_2)$ a general alignment the lists $l_1$ and $l_2$. Here $\delta_1$ and $\delta_2$ are strings over $\Sigma \cup \{\_\}$. Afterwards the similarity between $l_1$ and $l_2$ is defined as the greatest $\mathcal{E}((\delta_1, \delta_2))$ with $\mathcal{E}$ function with values in $\mathbf{N}$ that associate a score to each alignment $(\delta_1, \delta_2)$.

In computational biology the similarity of $l_1$ and $l_2$ is efficiently solved using dynamic programming; in fact the problem can be solved storing in a matrix $M$, of dimension $|l_1| \times |l_2|$, the values of the similarities between all the prefixes of length $i \leq |l_1|$ and $j \leq |l_2|$ of $l_1$ and $l_2$. This could be seen as a sort of generalization of the Fibonacci problem to 2-dimensions.

In this work we will formalize the *similarity* problem in the proof assistant Minlog. We will extract, from the proof of the existence of an alignment with highest score between two given strings the naive exponential program to compute the similarity of strings. Afterwards, we will propose a method to transform the given proof into another from which it will be possible to extract

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** : | R | I | M | D | M | D | M | M | I |
| **2** : | $v$ | _ | $i$ | $n$ | $t$ | $n$ | $e$ | $r$ | _ |
| **3** : | $w$ | $r$ | $i$ | _ | $t$ | _ | $e$ | $r$ | $s$ |

(6.1)

Figure 6.1: Alignment (lines 2, 3) and edit-transcript (line 1) of the strings *wintner* and *writers*. It is possible to note how the two methods are equivalent: a mismatch in the alignment correspond to a replacement in the edit transcript, a space in the alignment contained in the first string correspond to the insertion of the opposite character in first string, and a space in the alignment contained in the second string correspond to a deletion of the opposite character in the first string.

a more efficient program, in dynamic programming style.

We propose a method that we name *list as memory*. The idea consist in evaluating a sufficient amount of data in advance so that the extracted algorithm gets to reuse it instead of recomputing it each time it is needed. This is done by introducing in the proof a list of *ad-hoc* axioms. The method we propose cannot be applied automatically to an arbitrary proof; it can be seen more as a general schema (that has to be instantiated case by case) to follow in order extract dynamic programs from proofs.

This chapter is organized as follow: in section 6.1.1 we formalize the proof of the existence of an alignment with highest score between lists and we extract a program from the proof. The designed solution enumerate all the alignments in order to find the right one, and this generate an exponential running time extracted algorithm. In section 6.1.2 we present a proof transformation to apply to the proof presented in section 6.1.1 in order to extract an algorithm in dynamic programming style. In section 6.2, we make some final considerations over the presented method and future works.

### 6.1.1 The String Similarity Problem

Let $l_1$ and $l_2$ be two lists built on the alphabet $\Sigma$, with $\Sigma$ equal to $\mathbf{N}_{>0}$ (the set of naturals strictly higher than zero), $0 \notin \Sigma$ be the *space* character and $\alpha : \mathbf{N} \times \mathbf{N} \to \mathbf{Z}$ be the *scoring function* on a pair of symbols.

Given two lists $l_1$ and $l_2$ over $\Sigma$, in Figure 6.2 we give an inductive definition of the family of sets $\mathcal{A}_{i,j}^{l_1,l_2}$, the set of the *alignments* between the first $i \leq |l_1|$ characters of $l_1$ and $j \leq |l_2|$ characters of $l_2$.

In Figure 6.2 and in the rest of the chapter we make use of the following conventions: $n$, $m$, $i$ and $j$ ranges over $\mathbf{N}$, $|l|$ is the length of $l$, $l[i]$ is the $i+1$-th character of $l$, $\mathsf{head}(a :: l) = a$, $\mathsf{tail}(a :: l) = l$, $pre_n(l)$ is a partial operator

$$(A_0)\ \frac{}{((:),(:)) \in \mathcal{A}^{l_1,l_2}_{0,0}} \qquad (A_1)\ \frac{}{(0^{j+1}, pre_{j+1}(l_2)) \in \mathcal{A}^{l_1,l_2}_{0,j+1}}$$

$$(A_2)\ \frac{}{(pre_{i+1}(l_1), 0^{i+1}) \in \mathcal{A}^{l1,l2}_{i+1,0}} \qquad (A_3)\ \frac{(\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{i+1,j}}{(\delta_1 \cdot (0), \delta_2 \cdot l_2[j]) \in \mathcal{A}^{l_1,l_2}_{i+1,j+1}}$$

$$(A_4)\ \frac{(\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{i,j+1}}{(\delta_1 \cdot l_1[i], \delta_2 \cdot (0)) \in \mathcal{A}^{l_1,l_2}_{i+1,j+1}} \qquad (A_5)\ \frac{(\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{i,j}}{(\delta_1 \cdot l_1[i], \delta_2 \cdot l_2[j]) \in \mathcal{A}^{l_1,l_2}_{i+1,j+1}}$$

Figure 6.2: Induction definition of the alignments $\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}$

$$(E_0)\ \frac{}{\mathcal{E}[((:),(:))]_\alpha = 0} \qquad (E_1)\ \frac{}{\mathcal{E}[(0^j, pre_j(l_2))]_\alpha = \sum_{k=1}^{j} \alpha(0, l_2[k])}$$

$$(E_2)\ \frac{}{\mathcal{E}[(pre_i(l_1), 0^i)]_\alpha = \sum_{k=1}^{i} \alpha(l_1[k], 0)} \qquad (E_3)\ \frac{\mathcal{E}[(\delta_1,\delta_2)] = n}{\mathcal{E}[(\delta_1 \cdot (0), \delta_2 \cdot l_2[j])]_\alpha = n + \alpha(0, l_2[j])}$$

$$(E_4)\ \frac{\mathcal{E}[(\delta_1,\delta_2)] = n}{\mathcal{E}[(\delta_1 \cdot l_1[i], \delta_2 \cdot (0))]_\alpha = n + \alpha(l_1[i], 0)} \qquad (E_5)\ \frac{\mathcal{E}[(\delta_1,\delta_2)] = n}{\mathcal{E}[(\delta_1 \cdot l_1[i], \delta_2 \cdot l_2[j])]_\alpha = n + \alpha(l_1[i], l_2[j])}$$

Figure 6.3: Induction definition of the *evaluator* function $\mathcal{E}$

that return the first $n$ elements of a list $l$, $0^n$ is the list composed by a sequence of $n$ zeros, $l \cdot g$ the operation of appending the list $g$ to $l$ and $(a_1, \ldots, a_n)$ is the list composed by $a_i \in \mathbf{N}$.

We associate a *score* to each alignment by the *evaluator* function $\mathcal{E} : \mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|} \rightarrow (\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}) \rightarrow \mathbf{N}$ defined on the inductive structure of $\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}$ (Figure 6.3). The function $\mathcal{E}$ take as input an alignment, a scoring function and return the score of the input alignment. Our goal is to find the alignment in $\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}$ with highest score (this score will be the *similarity* between $l_1$ and $l_2$) with respect to a given scoring function $\alpha$.

Remark Many problems can be modeled as special case of similarity by choosing an appropriate scoring function $\alpha$. Let consider (below) the definition of the *longest common subsequence problem.*

**Definition 6.1.1.** *A subsequence of a string $l$ is specified by a list of indices*

*$i_1 < i_2 < \ldots < i_k$ for some $k \leq |l|$. The subsequence specified by this list is the string $l[i_1]l[i_2]\ldots l[i_k]$*

**Definition 6.1.2** (Longest Common Subsequence Problem)**.** *Given two strings $l_1$ and $l_2$ a common subsequence of $l_1$ and $l_2$ is a sequence that appear both in $l_1$ and $l_2$ as a subsequence. The Longest Common Subsequence Problem consist in finding the longest common subsequence between $l_1$ and $l_2$*

For example, 145 is a common subsequence of 114666725 and 1124375 but 11475 is the longest common ones. The solution of the longest common subsequence problem can be obtained from the solution of the similarity of lists problem by choosing a scoring function $\alpha$ that scores a "1" for each *match* and "0" for each *mismatch* or presence of a 0 (the result will depend by the implemented strategy to solve the problem since there could be more alignments with the same highest score).

Now we show formally that given a couple of lists $l_1, l_2$ over $\Sigma$ there exists always an alignment in $\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}$ of maximum score with respect to $\alpha$.

**Theorem 6.1.1.**
$\forall l_1, l_2 \exists \delta_1, \delta_2((\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}) \wedge$
$\forall \delta_1', \delta_2'((\delta_1', \delta_2') \in \mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|} \rightarrow \mathcal{E}(\delta_1', \delta_2') \leq \mathcal{E}(\delta_1, \delta_2))$

*Proof.* We assume $l_1$ and $l_2$. In order to prove the thesis we prove the following statement:
$\forall n, m \exists \delta_1, \delta_2((\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m}) \wedge$
$\forall \delta_1', \delta_2'((\delta_1', \delta_2') \in \mathcal{A}^{l_1,l_2}_{n,m}) \rightarrow \mathcal{E}(\delta_1', \delta_2') \leq \mathcal{E}(\delta_1, \delta_2)$
Obviously we obtain the thesis instantiating this assertion on $|l_1|$ and $|l_2|$. From now on we will write $Q(\delta_1, \delta_2, n, m)$ for

$$((\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m}) \wedge \forall \delta_1', \delta_2'((\delta_1', \delta_2') \in \mathcal{A}^{l_1,l_2}_{n,m}) \rightarrow \mathcal{E}(\delta_1', \delta_2') \leq \mathcal{E}(\delta_1, \delta_2)$$

We go by induction on $n$ and $m$.
**Base Case**[$n = 0$] We prove

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1, \delta_2, 0, m)$$

by case distinction over $m$:
**Base Case**[$n = 0$, $m = 0$]: $Q((:), (:), 0, 0)$ by rule $(A_0)$.
**Induction Step**[$n = 0$, $m + 1$]
We have $Q(0^{m+1}, pre_{m+1}(l_2), 0, m + 1)$ by rule $(A_1)$.

**Induction Step**[$n + 1$] We now assume

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n, m) \tag{6.2}$$

and we must show

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n+1, m)$$

By induction over $m$:

**Base Case**$[n+1,\ m=0]$
$Q(pre_{n+1}(l_1), 0^{n+1}, n+1, 0)$ by $(A_2)$

**Induction Step**$[n+1,\ m+1]$: Assume

$$\exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n+1, m) \tag{6.3}$$

we have to prove

$$\exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n+1, m+1)$$

By (6.3) there exists $\delta_1'$,$\delta_2'$ such that $(\delta_1', \delta_2') \in \mathcal{A}^{l_1,l_2}_{n+1,m}$ and such that for every $(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n+1,m}$

$$\mathcal{E}(\delta_1, \delta_2) \leq \mathcal{E}(\delta_1', \delta_2') \tag{6.4}$$

Instantiating (6.2) on $m+1$ there exists $\delta_1'', \delta_2''$ such that $(\delta_1'', \delta_2'') \in \mathcal{A}^{l_1,l_2}_{n,m+1}$ and for every $(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m+1}$

$$\mathcal{E}(\delta_1, \delta_2) \leq \mathcal{E}(\delta_1'', \delta_2'') \tag{6.5}$$

Instantiating (6.2) on $m$ there exists $\delta_1''', \delta_2'''$ such that $(\delta_1'', \delta_2'') \in \mathcal{A}^{l_1,l_2}_{n,m}$ and for every $(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m}$

$$\mathcal{E}(\delta_1, \delta_2) \leq \mathcal{E}(\delta_1''', \delta_2''') \tag{6.6}$$

Now we have to dispatch over the following cases:

$\mathbf{ip}_1$. $\underline{\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])}$:
Then, only 2 cases are possible:

$\mathbf{ip}_{1.1}$. $\underline{\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])}$: We claim $Q(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1], n+1, m+1)$. This is proved dispatching over $(\delta_1, \delta_2)$ in $\mathcal{A}^{l_1,l_2}_{n+1,m+1}$. In fact for every $(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n+1,m+1}$ only three cases are possible

$\mathbf{ip}_{1.1.1}\underline{(\delta_1, \delta_2) = (\delta_1^* \cdot (0:), \delta_2^* \cdot l_2[m+1])}$

$$\begin{array}{rcl} \mathcal{E}(\delta_1, \delta_2) & = & \mathcal{E}(\delta_1^* \cdot (0:), \delta_2^* \cdot l_2[m+1]) \\ & = & \mathcal{E}(\delta_1^*, \delta_2^*) + \alpha((0:), l_2[m+1]) \quad \text{by}(E_3) \\ & \leq & \mathcal{E}(\delta_1', \delta_2') + \alpha((0:), l_2[m+1]) \quad \text{by (6.4)} \end{array}$$

$$= \quad \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1]) \quad \text{by}(E_3)$$

$\mathbf{ip}_{1.1.2}$ $\underline{(\delta_1, \delta_2) = (\delta_1^* \cdot l_1[n+1]), \delta_2^* \cdot (0:))}$: So,

$$\begin{aligned}
\mathcal{E}[(\delta_1, \delta_2)] &= \mathcal{E}(\delta_1^* \cdot l_1[n+1], \delta_2^* \cdot (0:)) \\
&= \mathcal{E}(\delta_1^*, \delta_2^*) + \alpha(l_1[n+1], (0:)) \quad \text{by}(E_4) \\
&\leq \mathcal{E}(\delta_1'', \delta_2'') + \alpha(l_1[n+1], (0:)) \quad \text{by (6.5)} \\
&= \mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \\
&\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1]) \quad \text{by } (\mathbf{ip}_1)
\end{aligned}$$

$\mathbf{ip}_{1.1.3}$ $\underline{(\delta_1, \delta_2) = (\delta_1^* \cdot l_1[n+1]), \delta_2^* \cdot l_2[m+1])}$

$$\begin{aligned}
\mathcal{E}[(\delta_1, \delta_2)] &= \mathcal{E}(\delta_1^* \cdot l_1[n+1], \delta_2^* \cdot l_2[m+1]) \\
&= \mathcal{E}(\delta_1^*, \delta_2^*) + \alpha(l_1[n+1], l_2[m+1]) \quad \text{by}(E_5) \\
&\leq \mathcal{E}(\delta_1''', \delta_2''') + \alpha(l_1[n+1], l_2[m+1]) \quad \text{by (6.6)} \\
&= \mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \\
&\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1]) \quad \text{by } (\mathbf{ip}_{1.1})
\end{aligned}$$

$\mathbf{ip}_{1.2}$ $\underline{\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \not\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])}$: We claim $Q(\delta_1''' \cdot (l_1[n+1]), \delta_2''' \cdot l_2[m+1], n+1, m+1)$. The proof of this claim is done, as in the previous case, dispatching over $(\delta_1, \delta_2)$ in $\mathcal{A}_{n+1,m+1}^{l_1,l_2}$.

$\mathbf{ip}_2$ $\underline{\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \not\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])}$: Then there exists only two cases:

$\mathbf{ip}_{2.1}$ $\underline{\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \leq \mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:))}$: We claim $Q(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:), n+1, m+1)$.

$\mathbf{ip}_{2.2}$ $\underline{\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \not\leq \mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:))}$: We claim $Q(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1], n+1, m+1)$. The proofs of the previous two claims is done dispatching over $(\delta_1, \delta_2)$ in $\mathcal{A}_{n+1,m+1}^{l_1,l_2}$. □

The theorem 6.1.1 can be simply modified in order to construct not only the alignment with highest score but also the score itself (that is the *similarity*).
The program extracted from the previous proof is the following:

```
[l,g,alpha]
(Rec nat=>nat=>(list nat @@ list nat))
 ([m] if (m=0) ((:), (:))
               ((zeros (m+1)), (pre (m+1) g))
 ([n, f: (nat=>(list nat @@ list nat))]
  (Rec nat=>(list nat, list nat))
   ((pre (n+1) l), (zeros (n+1)))
   ([m,(d_1',d_2')]
     [LET (d_1'', d_2'') = (f (m+1)) IN
```

```
[LET (d_1''', d_2''') = (f m) IN
  [IF ((E (d_1'':+: l[n+1]) 0 alpha) <=
        (E (d_1':+: (:)) (d_2':+: g[m+1]) alpha))
   [IF ((E (d_1''':+: l[n+1]) (d_2''':+: g[m+1]) alpha)
         <=(E (d_1':+: (:)) (d_2':+: g[m+1]) alpha))
     ((d_1':+:(:)), (d_2':+: g[m+1]))
     ((d_1''':+: l[n+1]) (d_2''':+: g[m+1]))]
   [IF ((E (d_1''':+: l[n+1]) (d_2''':+: g[m+1]) alpha)
         <=(E (d_1'':+: l[n+1]) (d_2'':+:  (:)) alpha))
     ((d_1'':+: l[n+1]) (d_2'' :+: (:)))
     ((d_1''':+:l[n+1]) (d_2''':+:g[m+1]))]]]]))|l||g|
```

Here we indicated by `(pre n)` the operator $pre_n$, by `(zeros n)` the string $0^n$, by `E` the function $\mathcal{E}$ and by `alpha` the scoring function $\alpha$.

**Complexity of the Extracted Algorithm**: The complexity of the extracted program can be modeled by the following recurrence:

$$T_1(n,m) = \begin{cases} k_1 m & n = 0 \\ T_2(m) & n > 0 \end{cases}$$

with

$$T_2(m) = \begin{cases} k_2 n & m = 0 \\ T_2(m-1) + T_1(n-1,m) + \\ T_1(n-1,m-1) + 2k_3 \mathsf{max}(n+m) & m > 0 \end{cases}$$

with $2k_3\mathsf{max}(n+m)$ cost for the severals application of the append operation in the body of the nested recursion. The complexity of the extracted program then will be given by $T_1(|l_1|,|l_2|)$. Given $n > 0$ and $m > 0$ the unfolding of $T_1(n,m)$ can be represented as a ternary tree where the lowest branch has high $m$ and the highest $n+m$. Thus the extracted programs has a number of recursive calls in $\Omega(3^{\min(n,m)})$.

### 6.1.2 List as Memory Paradigm

To drastically reduce the complexity of our extracted program, we developed a method that we named *list as memory*. The idea consist in evaluating a sufficient amount of data in advance so that the extracted algorithm gets to reuse it instead of recomputing it each time it is needed.

The basic idea is still to prove Theorem6.1.1 by a double induction (before on the length $|l_1|$ of the first list and by a nested induction on $|l_2|$ length of the second list) but this time using an additional data structure $w$, a FIFO (*First In First Out*) list where we store the alignments with highest score computed in the previous steps. The list $w$ will be built and updated during the proof

and it will constitute part of the witness of the new proof together with the alignment of highest score.

Thus assuming we want to compute the best alignment of the first $n+1$ characters of $l_1$ and $m+1$ character of $l_2$, we will assume $w$ to be the following list of alignments:

$$(\delta_1,\delta_2)^{l_1,l_2}_{n,m},(\delta_1,\delta_2)^{l_1,l_2}_{n,m+1},\dots(\delta_1,\delta_2)^{l_1,l_2}_{n,|l_2|}$$
$$(\delta_1,\delta_2)^{l_1,l_2}_{n+1,0},(\delta_1,\delta_2)^{l_1,l_2}_{n+1,1}\dots,(\delta_1,\delta_2)^{l_1,l_2}_{n+1,m}$$

with $(\delta_1,\delta_2)^{l_1,l_2}_{i,j}$ alignment of highest score between the first $i$ characters of $l_1$ and $j$ characters of $l_2$. At this point the intended alignment will be computed considering the head of $w$, $(\delta_1,\delta_2)^{l_1,l_2}_{n,m}$, the head of the tail of $w$, $(\delta_1,\delta_2)^{l_1,l_2}_{n,m+1}$ and the recursive call of the nested induction on $l_2$ (the alignment of highest score between the first $n+1$ element of $l_1$ and $m$ elements of $l_2$, that here occur as last element in $w$) Once the new alignment is computed the list $w$ has to be properly updated.

So in general the idea is to replace the double instantiation of the induction hypothesis (6.2) in Theorem 6.1.1 (that correspond to the two recursive calls in the extracted algorithm) with just a reading operation of the *head* and the *head* of the *tail* of our *memory* list $w$.

In order to use such memory list in our proof we have to modify the original proof of the Theorem 6.1.1 in an appropriate way. More precisely we introduce the predicate $\mathtt{MEM} \subseteq \mathbf{L}(\mathbf{N}_{>0}) \times \mathbf{L}(\mathbf{N}_{>0}) \times \mathbf{N} \times \mathbf{N} \times \mathbf{L}(\mathbf{L}(N) \times \mathbf{L}(N))$ where,

- $(\mathtt{MEM}\ l_1\, l_2\, 0\, v\, w)$, stands for "in $w$ are stored the the $v+1$ alignments $(0^k, pre_k(l_2))$ with $k = 0,\dots v$" (here we assume $0^0 = (:)$ and $pre_0(l_2) = (:)$) and

- $(\mathtt{MEM}\ l_1\, l_2\, (u+1)\, v\, w)$, stands for "in $w$ are stored the $|l_2|+2$ alignments of highest scores between the first $j$ and $k$ characters of $l_1$ and $l_2$ with

$$(j,k) \in \{(u,v),\dots,(u,|l_2|),(u+1,0),\dots,(u+1,v)\}$$

and the following set of axioms specifying the necessary operations to build and correctly update the memory list $w$:

**[I]**(*Initialization*),

$$\forall l_1, l_2, m, w(\mathtt{MEM}\ l_1\, l_2\, 0\, m\, (\mathsf{init}\, m\, l_2))$$

with

$$(\mathsf{init}\, m\, l_2) = \begin{cases} ((:),(:)) & m = 0 \\ ((\mathsf{init}\,(m-1)) :+: (0^m, pre_m\, l_2)) & 0 < m \end{cases}$$

**[H]** (*Head of the list*):

$\forall l_!, l_2, n, m, w(\texttt{MEM}\ l_1\, l_2\, (n+1)\, m\, w) \rightarrow$
$Q(\pi_0(\mathsf{head} w), \pi_1(\mathsf{head} w), n, m)$

**[HT]** (*Head of the tail*):
$\forall l_!, l_2, n, m, w(m < |l_2|) \rightarrow (\texttt{MEM}\ l_1\, l_2\, (n+1)\, m\, w) \rightarrow$
$Q(\pi_0(\mathsf{head}(\mathsf{tail} w)), \pi_1(\mathsf{head}(\mathsf{tail} w)), n, m+1)$

**[CL]** (*Change Line*):
$\forall l_1, l_2, n, m, w(\texttt{MEM}\ l_1\, l_2\, n\, |l_2|\, w) \rightarrow$
$(\texttt{MEM}\ l_1\, l_2\, (n+1)\, 0\, ((\mathsf{tail} w) :+: ((pre_{n+1}\, l_1), 0^{n+1}))$

**[OSOR1]** (*One Step On the Right 1*):
$\forall l_1, l_2, n, m, \delta_1', \delta_2', w(m < |l_2|) \rightarrow (Q\, \delta_1'\, \delta_2'\, n\, m) \rightarrow$
$(\texttt{MEM}\ l_1\, l_2\, n\, m\, w) \rightarrow$
$(\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])) \rightarrow$
$(\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])) \rightarrow$
$(\texttt{MEM}\ l_1\, l_2\, n\, (m+1)\, ((\mathsf{tail} w) \cdot (\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])))$
with $(\delta_1''', \delta_2''') = (\mathsf{head}\, w)$ and $(\delta_1'', \delta_2'') = (\mathsf{head}\, (\mathsf{tail}\, w))$.

**[OSOR2]** (*One Step On the Right 2*):
$\forall l_1, l_2, n, m, \delta_1', \delta_2', w(m < |l_2|) \rightarrow (Q\, \delta_1'\, \delta_2'\, n\, m) \rightarrow$
$(\texttt{MEM}\ l_1\, l_2\, n\, m\, w) \rightarrow$
$(\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])) \rightarrow$
$(\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \not\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])) \rightarrow$
$(\texttt{MEM}\ l_1\, l_2\, n\, (m+1)\, ((\mathsf{tail} w) \cdot (\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1])))$
with $(\delta_1''', \delta_2''') = (\mathsf{head}\, w)$ and $(\delta_1'', \delta_2'') = (\mathsf{head}\, (\mathsf{tail}\, w))$.

**[OSOR3]** (*One Step On the Right 3*):
$\forall l_1, l_2, n, m, \delta_1', \delta_2', w(m < |l_2|) \rightarrow (Q\, \delta_1'\, \delta_2'\, n\, m) \rightarrow$
$(\texttt{MEM}\ l_1\, l_2\, n\, m\, w) \rightarrow$
$(\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \not\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])) \rightarrow$
$(\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \leq (\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:))) \rightarrow$
$(\texttt{MEM}\ l_1\, l_2\, n\, (m+1)\, ((\mathsf{tail} w) \cdot (\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:))))$
with $(\delta_1''', \delta_2''') = (\mathsf{head}\, w)$ and $(\delta_1'', \delta_2'') = (\mathsf{head}\, (\mathsf{tail}\, w))$.

**[OSOR4]** (*One Step On the Right 4*):
$\forall l_1, l_2, n, m, \delta_1', \delta_2', w(m < |l_2|) \rightarrow (Q\, \delta_1'\, \delta_2'\, n\, m) \rightarrow$
$(\texttt{MEM}\ l_1\, l_2\, n\, m\, w) \rightarrow$
$(\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \not\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])) \rightarrow$
$(\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \not\leq (\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:))) \rightarrow$
$(\texttt{MEM}\ l_1\, l_2\, n\, (m+1)\, ((\mathsf{tail} w) \cdot (\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1])))$
with $(\delta_1''', \delta_2''') = (\mathsf{head}\, w)$ and $(\delta_1'', \delta_2'') = (\mathsf{head}\, (\mathsf{tail}\, w))$.

**Theorem 6.1.2.** ***[I]*** $\rightarrow$ ***[CL]*** $\rightarrow$ ***[H]*** $\rightarrow$ ***[HT]*** $\rightarrow$ ***[OSOR1]*** $\rightarrow$ ***[OSOR2]*** $\rightarrow$ ***[OSOR3]*** $\rightarrow$ ***[OSOR4]*** $\rightarrow \forall l_1, l_2(\exists \delta_1, \delta_2(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|} \wedge \forall \delta_1', \delta_2'((\delta_1', \delta_2') \in$

$$\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|} \rightarrow \mathcal{E}(\delta_1',\delta_2') \leq \mathcal{E}(\delta_1,\delta_2))) \wedge \exists w(\texttt{MEM}\ l_1\, l_2\, |l_1|\, |l_2|\, w)$$

*Sketch.* Assume **[I]**, **[CL]**, **[H]**, **[HT]**, **[OSOR1]**, **[OSOR2]**, **[OSOR3]**, **[OSOR4]** $l_1$ and $l_2$. In order to prove the theorem 6.1.2 we prove the following assertion:

$\forall n, m(\exists \delta_1, \delta_2(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m} \wedge$
$\forall \delta_1', \delta_2'(\delta_1', \delta_2') \in \mathcal{A}^{l_1,l_2}_{n,m} \rightarrow \mathcal{E}(\delta_1', \delta_2') \leq \mathcal{E}(\delta_1, \delta_2)) \wedge$
$\exists w(\texttt{MEM}\ l_1\, l_2\, n\, m\, w)$

By induction on $n$ and $m$.

**Base Case**$[n = 0]$ We prove

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1, \delta_2, 0, m) \wedge \exists w(\texttt{MEM}\ l_1\, l_2\, 0\, m\, w)$$

by case distincion over $m$:

**Base Case**$[n = 0,\ m = 0]$
$Q((:),(:),0,0) \wedge \exists w(\texttt{MEM}\ l_1\, l_2\, 0\, 0\, (\texttt{init}\, 0\, l_2)))$ by rule $(A_0)$ and **[I]**.

**Induction Step**$[n = 0,\ m + 1]$
We have $Q(0^{m+1}, pre_{m+1}(l_2), 0, m + 1) \wedge$
$\exists w(\texttt{MEM}\ l_1\, l_2\, 0\, (m+1)\, (\texttt{init}\, (m+1)\, l_2))$ by rule $(A_1)$ and **[I]**.

**Induction Step**$[n + 1]$
We now assume

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n, m) \wedge \exists w(\texttt{MEM}\ l_1\, l_2\, n\, m\, w) \tag{6.7}$$

and we show

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n+1, m) \wedge \exists w(\texttt{MEM}\ l_1\, l_2\, (n+1)\, m\, w)$$

By induction over $m$:

**Base Case**$[n + 1,\ m = 0]$ $Q(pre_{n+1}(l_1), 0^{n+1}, n + 1, 0)$ by $(A_2)$. Then instantiating (6.7) on $|l_2|$ we have $\overline{w}$ such that $\exists w(\texttt{MEM}\ l_1\, l_2\, n\, |l_2|\, \overline{w})$ and by **[CL]** we have $(\texttt{MEM}\ l_1\, l_2\, (n+1)\, 0\, ((\texttt{tail}\, \overline{w}) \cdot ((pre_{n+1}\, l_1), 0^{n+1}))$.

**Induction Step**$[n + 1,\ m + 1]$ Assume

$$\exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n+1, m) \wedge \exists w(\texttt{MEM}\ l_1\, l_2\, (n+1)\, m\, w) \tag{6.8}$$

we prove

$$\exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n+1, m+1) \wedge \exists w(\texttt{MEM}\ l_1\, l_2\, (n+1)\, (m+1)\, w)$$

By (6.8) there exists $\delta_1'$,$\delta_2'$ such that $(\delta_1', \delta_2') \in \mathcal{A}^{l_1,l_2}_{n+1,m}$ and such that for every $(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n+1,m}$

$$\mathcal{E}(\delta_1, \delta_2) \leq \mathcal{E}(\delta_1', \delta_2') \tag{6.9}$$

By (6.8) let $\overline{w}$ be such that $(\texttt{MEM}\ l_1\, l_2\, (n+1)\, m\, \overline{w})$. By **[HT]**, we have that $(\delta_1'', \delta_2'') \in \mathcal{A}_{n,m+1}^{l_1,l_2}$ and for every $(\delta_1, \delta_2) \in \mathcal{A}_{n,m+1}^{l_1,l_2}$

$$\mathcal{E}(\delta_1, \delta_2) \leq \mathcal{E}(\delta_1'', \delta_2'') \tag{6.10}$$

with $(\delta_1'', \delta_2'') = (\mathsf{head}(\mathsf{tail}\,\overline{w}))$.

By **[H]** we have that that $(\delta_1''', \delta_2''') \in \mathcal{A}_{n,m}^{l_1,l_2}$ and for every $(\delta_1, \delta_2) \in \mathcal{A}_{n,m}^{l_1,l_2}$

$$\mathcal{E}(\delta_1, \delta_2) \leq \mathcal{E}(\delta_1''', \delta_2''') \tag{6.11}$$

with $(\delta_1'', \delta_2'') = (\mathsf{head}\,\overline{w})$. Now we have to dispatch over the following cases:

$\mathsf{ip}_1$. $\underline{\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])}$: Then, only 2 cases are possible:

$\mathsf{ip}_{1.1}$. $\underline{\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])}$: We claim

$$Q(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1], n+1, m+1)$$

and

$$(\texttt{MEM}\ l_1\, l_2\, n\, (m+1)\, ((\mathsf{tail}\, w) \cdot (\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])))$$

This is proved dispatching over $(\delta_1, \delta_2)$ in $\mathcal{A}_{n+1,m+1}^{l_1,l_2}$ and by **[OSOR1]**, $\mathsf{ip}_1$ and $\mathsf{ip}_{1.1}$.

$\mathsf{ip}_{1.2}$. $\underline{\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \not\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])}$: We claim

$$Q(\delta_1''' \cdot (l_1[n+1]), \delta_2''' \cdot l_2[m+1], n+1, m+1)$$

and

$$(\texttt{MEM}\ l_1\, l_2\, n\, (m+1)\, ((\mathsf{tail}\, w) \cdot (\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1])))$$

This is proved dispatching over $(\delta_1, \delta_2)$ in $\mathcal{A}_{n+1,m+1}^{l_1,l_2}$ and by **[OSOR2]**, $\mathsf{ip}_1$ and $\mathsf{ip}_{1.2}$

$\mathsf{ip}_2$. $\underline{\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \not\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])}$:Then there exists only two cases:

$\mathsf{ip}_{2.1}$. $\underline{\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \leq \mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:))}$: We claim

$$Q(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:), n+1, m+1)$$

and

$$(\texttt{MEM}\ l_1\, l_2\, n\, (m+1)\, ((\mathsf{tail}\, w) \cdot (\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)))$$

Proved dispatching over $(\delta_1, \delta_2)$ in $\mathcal{A}_{n+1,m+1}^{l_1,l_2}$ and by **[OSOR3]**, $\mathsf{ip}_2$ and $\mathsf{ip}_{2.1}$

$\mathsf{ip}_{2.2}$. $\underline{\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \not\leq \mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:))}$: We claim

$$Q(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1], n+1, m+1)$$

and

$$(\texttt{MEM}\ l_1\, l_2\, n\, (m+1)\, ((\texttt{tail}\, w) \cdot (\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]))$$

Proved dispatching over $(\delta_1, \delta_2)$ in $\mathcal{A}^{l_1,l_2}_{n+1,m+1}$ and by proved by **[OSOR4]**, $\mathsf{ip}_2$ and $\mathsf{ip}_{2.2}$ □

From the previous proof we can extract the following program:

```
[l,g,alpha]
(Rec nat=>nat=>((list nat@@list nat)@@
                  (list(list nat@@list nat))
 ([m] [if (m=0)
      (((:),(:)) , ((:),(:)))
      (((nZeros (m+1)), (nPrefix (m+1) g)),(init (m+1) g)


([n,f:(nat=>((list nat@@list nat)@@
                          (list(list nat@@list nat)))]
  (Rec nat=>((list nat@@list nat)@@
                           (list(list nat@@list nat)))
   LET w = (f |g|) IN
    (((nPrefix (n+1) l), (nZeros (n+1))) ,
       ((tail w):+:((nPrefix (n+1) l),(nZeros(n+1)))))
    ([m,((d_1',d_2'),w)]
      [LET (d_1'', d_2'') = (head (tail w)) IN
       [LET (d_1''', d_2''') = (head w) IN
         [IF((E (d_1'':+: l[n+1]) 0 alpha) <=
              (E (d_1':+: (:)) (d_2':+: g[m+1]) alpha))
          [IF((E(d_1''':+: l[n+1])(d_2''':+: g[m+1])alpha)
               <=(E (d_1':+: (:))(d_2':+: g[m+1]) alpha))
           (((d_1':+:(:)),(d_2':+:g[m+1])),
             ((tail w):+:(d_1':+:(:)),(d_2':+:g[m+1])))
           (((d_1''':+: l[n+1])(d_2''':+:g[m+1])),
            ((tail w):+:(d_1''':+: l[n+1])(d_2''':+:g[m+1])))]
          [IF((E(d_1''':+: l[n+1]) (d_2''':+: g[m+1])alpha)
               <= (E(d_1'':+:l[n+1]) (d_2'':+: (:))alpha))
           (((d_1'':+: l[n+1])(d_2'' :+: (:))),
             ((tail w):+:((d_1'':+: l[n+1]) (d_2'':+:(:)))))
           (((d_1''':+: l[n+1]) (d_2''':+: g[m+1])),
            ((tail w):+:((d_1''':+:l[n+1]) (d_2''':+:g[m+1]))))
           ]]]]))|l| |g|
```

#### 6.1.2.1 Complexity Considerations

The complexity of the extracted program can be modeled by the following recurrence (here we have as additional parameter the length of $g$):

$$T_1(n,m) = \begin{cases} k_1 m & n = 0 \\ T_2(n,m) & n > 0 \end{cases}$$

with

$$T_2(n,m) = \begin{cases} T_1(n-1,|g|) & m = 0 \\ T_2(n,m-1) + 2k_3 \mathsf{max}(n+m) & m > 0 \end{cases}$$

Given $|l| > 0$ and $|g| > 0$ the unfolding of $T_1(|l|,|g|)$ can be represented by the following $|l| \times |g|$ matrix of list of calls:

$$\begin{array}{lllll} & T_1(|l|,|g|) & \rightarrow T_2(|l|,|g|) & \rightarrow \ldots \rightarrow & T_2(|l|,0) \\ \rightarrow & T_1(|l|-1,|g|) & \rightarrow T_2(|l|-1,|g|) & \rightarrow \ldots \rightarrow & T_2(|l|-1,0) \\ & & \vdots & & \\ \rightarrow & T_1(1,|g|) & \rightarrow T_2(1,|g|) & \rightarrow \ldots \rightarrow & T_2(1,0) \end{array}$$

and being the complexity of each call $2k_3 \mathsf{max}(|l| + |g|)$ then $T_1(|l|,|g|)$ is in $\mathcal{O}(|l||g|\mathsf{max}(|l||g|))$

## 6.2 Conclusions

With an opportune modification of the alignment definition in Figure 6.2 we can avoid the cost relative to the applications of the append function. In this way, the extracted program from the efficient implementation of the existence of an alignment with highest score will have a complexity in $\mathcal{O}(|l||g|)$. Future work will regards a sort of automation of the presented method.

# 7 Tail Recursion

## 7.1 Introduction

Let $M$ be a proof by induction over $n$ (natural number) of the property $\forall nA(n)$, and let, by the *Proofs-as-Program* paradigm, $[\![M]\!]$ be the (recursive) content of $M$. In this chapter we will try to answer the following question: *How to turn automatically $M$ into another proof, say $N$, with tail recursive content?* Penny Anderson in her Phd thesis [1] used Frank Pfenning's *Insertion Lemma* [30] proof transformation, in order to extract *tail recursive* programs from proofs. This method, although particularly interesting, is *user dependent*. What we will do here is to present and develop in a formal setting an idea first roughly introduced in [9] (originated from an informal chat the author had with Andrej Bauer in 2004, reported in the Bauer's mathematical blog[1]) in order to extract tail recursive programs from proofs but in a completely automatic fashion.

Let us consider the following program, written in an ML-like syntax:

```
let rec FACT n = if n = 0 then 1 else n * FACT (n - 1)
```

`FACT` computes the factorial of $n$, for any positive integer $n$. But this implementation is not *tail* recursive because in each step of the computation the compiler has to store (on a stack) the context $(n * [])$, evaluate `FACT (n-1)` $\mapsto v$, and returns $(n * v)$. It is well known that `FACT` can be turned into a simpler function where it is not necessary to stack any context information:

```
let rec FACT' n =
  let rec FACT'' n m y =
    if n = 0 then y else FACT'' (n - 1) (m + 1) ((m + 1) * y)
  in FACT'' n 0 1
```

Now assume `FACT` to be the computational content of the proof by induction $M$, with end formula $\forall nA(n)$, that states that for each natural $n$ there exists $n!$. From which proof is it possible to extract `FACT'`? Both programs `FACT` and `FACT'` compute the factorial function, so `FACT'` should be the content of an appropriate proof of $\forall nA(n)$ as well. So the problem is shifted in understanding which logical property `FACT''` has. Given a natural $n$, (`FACT''`$n$) is a function

[1] `http://math.andrej.com/2005/09/16/proof-hacking/`

that takes the natural $m$, the witness $y$ for $A(m)$ and returns a witness for $A(n+m)$.

Hence given $n$, (FACT''$n\,0\,1$) is the witness for $A(n)$ as expected. Intuitively, we expect FACT'' to be the computational content of some proof of the formula $\forall n, m(A(m) \to A(n+m))$

Will show that this is the right intuition to follow for the automatic generation of tail recursive programs.

This chapter is organized as follows. In section 7.2 we address two proof transformations in order to extract *continuation* and *accumulator* based tail recursive programs, in section 7.3 we show that there exists a formal connection between the two proof transformations presented in section 7.2 and finally, in section 7.4 we apply our methods to a well known problem in bioinformatics, the *Maximal Scoring Subsequence Problem*.

## 7.2 Proof Manipulation

This section is devoted to expose the proofs transformation we have in mind in order to generate (by extraction) more efficient programs starting with a given inductive proof on natural numbers. How the techniques can be extended to other data types is discussed in the conclusion.

**Definition 7.2.1** (Tail Expressions [22]). *The tail expressions of $t \in Terms$, are defined inductively as follows:*

1. *If $t \equiv (\lambda x.e)$ then $e$ is a tail expression.*
2. *If $t \equiv (\mathit{if}\, t\, r\, s)$ is a tail expression, then both $r$ and $s$ are tail expressions.*
3. *If $t \equiv (\mathcal{R}_\iota\, r\, s)$ is a tail expression, then $r$ and $s$ are tail expressions.*
4. *Nothing else is a tail expression,*

*where $\iota \in \{\mathbf{N}, \boldsymbol{L}(\rho)\}$.*

**Definition 7.2.2.** *A tail call is a tail expression that is a procedure call.*

**Definition 7.2.3** (Tail Recursion [23]). *A recursive procedure is said to be tail recursive when it tail calls itself or calls itself indirectly through a series of tail calls.*

Now, let $F$ be the following induction proof over $\mathbf{N}$:

$$\frac{\begin{array}{cc} |M & |N \\ A(0) & \forall n(A(n) \to A(n+1)) \end{array}}{\forall n A(n)}$$

The content of $F$ is $(\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$ with $b$ and $f$ content of the proofs $M$ and $N$.

### 7.2.1 *Continuation* Based Tail Recursion

Given the procedure $(\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$ defined in the previous section, let $\Lambda$ be the term:

$$\mathcal{R}^{(\sigma\to\sigma')\to\sigma'}_{\mathbf{N}}\,(\lambda k.kb)(\lambda n,p,k.\,p\,\lambda u.k(f\,n\,u))$$

In $\Lambda$, the first input parameter, which has type $(\sigma \to \sigma')$. is called a *continuation*; $\Lambda$ is a function with just one tail recursive call and a functional accumulator parameter $k$ with the following property: for each $n$, at the $i$-th $(0 < i \leq n)$ step of the computation of $(\Lambda\, n\, (\lambda x.x))$ the continuation has the form $\lambda u.(f\,(n-1)\,(\ldots(f\,(n-i)\,u)\ldots))$. At the $n$-th step the continuation $\lambda u.(f\,(n-1)\,(\ldots(f\,0\,u)\ldots))$ is applied to the term $b$ and returns. We see that such returned value corresponds to $(\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)n$. This fact is stated formally in the following,

**Theorem 7.2.1.** *For each natural $n$:*

$$\Lambda\, n =_{\mathcal{R}\eta\beta} \lambda k^{\sigma\to\sigma'}.\, k((\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)n)$$

*Proof.* By induction over $n$:

$n = 0$

$$\begin{array}{lll} \Lambda\, 0 & =_{\mathcal{R}\eta\beta} & \lambda k.kb \\ & =_{\mathcal{R}\eta\beta} & \lambda k.\, k((\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)0) \end{array}$$

$n + 1$

$$\begin{array}{lll} \Lambda\,(n+1) & =_{\mathcal{R}\eta\beta} & (\lambda n,p,k.\,p\,\lambda u.k(f\,n\,u))\,n\,(\Lambda\, n) \\ & =_{\mathcal{R}\eta\beta} & \lambda k.(\Lambda\, n)\lambda u.k(f\,n\,u) \\ & =_{\mathcal{R}\eta\beta} & \lambda k.(\lambda k.k((\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)n))\lambda u.k(f\,n\,u) \quad \text{(by IH)} \\ & =_{\mathcal{R}\eta\beta} & \lambda k.(\lambda u.k(f\,n\,u))((\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)n) \\ & =_{\mathcal{R}\eta\beta} & \lambda k.k(f\,n\,((\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)n)) \\ & =_{\mathcal{R}\eta\beta} & \lambda k.k((\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)(n+1)) \end{array}$$

□

Now, as expected, when applied to the *identity continuation* $\lambda x.x$ we get another program in the same equivalence class:

**Corollary 7.2.2.** $\lambda n.\Lambda\, n\,(\lambda x.x) =_{\mathcal{R}\eta\beta} (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$

So we have at hand a *better* program. We still need to ensure it can be reached, in an automatic way, from another proof of the same given statement.

More formally, assume we are given some proof term $F$, with extraction $[\![F]\!] = (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$, is it possible to find out *another* proof $F'$ of the same statement, which leads to the other program: $[\![F']\!] = (\lambda n.\Lambda\, n\, (\lambda x.x))$. This is the challenge that we will answer positively below.

The key point is to understand the logical role of the continuation parameter in $\Lambda$: given a natural $n$, at each step $i : n, \ldots, 0$ in computing $(\Lambda\, n\, (\lambda x.x))$, the continuation is a function that takes the witness for $A(i)$ and returns the witness for $A(i+m)$, for $m$ such that $i + m = n$. So we expect $\Lambda$ to be the computational content of a proof with end formula:

$$\forall n \forall^{\mathsf{nc}} m((A(n) \to A(n+m)) \to A(n+m)) \tag{7.1}$$

We observe that the counter $m$ is introduced to count how much $n$ is *decreasing* during the computation. So, as such, it plays a "logical" role (or commentary role if one prefers); in other words, it is irrelevant at the programming level, and should be marked to be dropped out. To this end, we explicitly underline the "hidden" role of $m$ quantifying over it by the special *non-computational* quantifier $\forall^{\mathsf{nc}}$[5][4]. Let us prove the above statement (7.1), under the assumptions we have proofs for both $A(0)$ and $\forall n(A(n) \to A(n+1))$,

**Proposition 7.2.3.** $A(0) \to \forall n(A(n) \to A(n+1)) \to \forall n \forall^{nc} m((A(n) \to A(n+m)) \to A(n+m))$

*Proof.* Assume $b : A(0)$ and $f : \forall n(A(n) \to A(n+1))$. By induction on $n$.

$n = 0$ We have to prove

$$\forall^{\mathsf{nc}} m((A(0) \to A(m)) \to A(m))$$

So assume $m$ and $k : (A(0) \to A(m))$. Apply $k$ to $b : A(0)$.

$n+1$ Assume $n$, the recursive call $p : \forall^{\mathsf{nc}} m((A(n) \to A(n+m)) \to A(n+m))$, $m$, and the continuation $k : A(n+1) \to A(n+m+1)$. We have to prove:

$$A(n+m+1)$$

Apply $p$ to $(m+1)$ obtaining $(p\,(m+1)) : (A(n) \to A(n+m+1)) \to A(n+m+1)$. So, if we are able to prove the formula $A(n) \to A(n+m+1)$, by some proof $t$, we can just apply $(p\,(m+1))$ to $t$ and we are done.

So let us prove

$$A(n) \to A(n+m+1)$$

Assume $v : A(n)$. We apply $k$ to $(f\, n\, v)$.

□

**Proposition 7.2.4.** $A(0) \to \forall n(A(n) \to A(n+1)) \to \forall n A(n)$.

*Proof.* Assume $b : A(0)$, $f : \forall n(A(n) \to A(n+1))$. Given $n$, to prove $A(n)$, we instantiate the formula proved in Proposition 7.2.3 on $b$, $f$, $n$, 0 and $A(n) \to A(n)$. □

The content of the previous proof, that we name Ind_CONT, is the following:

```
[b,f,n] (Rec nat => (sigma => sigma) => sigma)
         ([k](k b))
         ([n,p,k] p ([u] k (f n u)) n ([x]x))
```

Notice that, although the functional parameter in $\Lambda$ is a *continuation*, $\Lambda$ is not of the kind provided alongside a CPS-transformation of the recursion over naturals schema. In fact $f$ and $b$ are not altered in our transformation and they could contain *bad* expressions, like not tail calls.

The formula (7.1) could be substituted by the more general $\forall n(A(n) \to \bot) \to \bot$. By an opportune adaptation of the proof of Proposition 7.2.3 we would have obtained the same computational content (of Porposition 7.2.4) Ind_CONT. However, here we offer a clearer formulation for the logical property the continuation parameter is supposed to satisfy. In addition, this approach represents a non trivial usage of the *non* computational quantifiers $\forall^{\mathsf{nc}}$.

### 7.2.2 *Accumulator* Based Tail Recursion

Here we present the essence of Bauer's [? ] original idea. Given the procedure $(\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$ defined in the last section, let $\Pi$ be the term:

$$\mathcal{R}^{\mathbf{N}\to\sigma\to\sigma}_{\mathbf{N}}\,(\lambda m, y.y)\,(\lambda n, p, m, y.\, p\,(m+1)\,(f\,m\,y))$$

In $\Pi$ there are two accumulator parameters: a natural and parameter of type $\sigma$ where intermediate results are stored. For each natural $n$, at the $i$-th $(0 < i \leq n)$ step of the computation of $(\Pi\, n\, 0\, b)$ the accumulator of the partial results will be equal to the expression $(f\,(i-1)\,(\ldots(f\,0\,b)\ldots))$. At the $n$-th step (base case of $\Pi$) the accumulator of the partial results is returned and it corresponds to $(\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)n$. This fact is stated in theorem 7.2.6 below.

**Definition 7.2.4.** *For all n,m, let* $\overline{f}^{\mathbf{N}\to\mathbf{N}\to\sigma\to\sigma}$ *be a function such that:*

$$\overline{f}_m\, n = f\,(n+m)$$

**Proposition 7.2.5.** *For all naturals $n$ and $m$:*

$$(\mathcal{R}^{\sigma}_{\mathbf{N}}\,(\overline{f}_m\, 0\, b)\,\overline{f}_{m+1})\, n =_{\mathcal{R}\eta\beta} (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\,\overline{f}_m)\,(n+1)$$

*Proof.* By induction on $n$.

$n = 0$

$$\begin{aligned}(\mathcal{R}^{\sigma}_{\mathbf{N}}\,(\overline{f}_m\,0\,b)\,\overline{f}_{m+1})\,0 &=_{\mathcal{R}\eta\beta} (\overline{f}_m\,0\,b)\\ &=_{\mathcal{R}\eta\beta} (\mathcal{R}^{\sigma}_{\mathbf{N}}\,b\,\overline{f}_m)\,1\end{aligned}$$

$n + 1$

$$\begin{aligned}(\mathcal{R}^{\sigma}_{\mathbf{N}}\,(\overline{f}_m\,0\,b)\,\overline{f}_{m+1})\,n+1 &=_{\mathcal{R}\eta\beta} \overline{f}_{m+1}\,n\,((\mathcal{R}^{\sigma}_{\mathbf{N}}\,(\overline{f}_m\,0\,b)\,\overline{f}_{m+1})\,n)\\ &=_{\mathcal{R}\eta\beta} \overline{f}_{m+1}\,n\,((\mathcal{R}^{\sigma}_{\mathbf{N}}\,b\,\overline{f}_m)\,(n+1)) \qquad \text{by IH}\\ &=_{\mathcal{R}\eta\beta} f\,(m+1+n)\,((\mathcal{R}^{\sigma}_{\mathbf{N}}\,b\,\overline{f}_m)\,(n+1))\\ &\qquad\qquad \text{by Def. 7.2.4}\\ &=_{\mathcal{R}\eta\beta} \overline{f}_m\,(n+1)\,((\mathcal{R}^{\sigma}_{\mathbf{N}}\,b\,\overline{f}_m)\,(n+1))\\ &=_{\mathcal{R}\eta\beta} (\mathcal{R}^{\sigma}_{\mathbf{N}}\,b\,\overline{f}_m)\,(n+2)\end{aligned}$$

□

**Theorem 7.2.6.** *For all naturals $n$,*

$$\Pi\,n =_{\mathcal{R}\eta\beta} \lambda m, y(\mathcal{R}^{\sigma}_{\mathbf{N}}\,y\,\overline{f}_m)\,n$$

*Proof.* By induction on $n$:

$n = 0$

$$\begin{aligned}\Pi\,0 &=_{\mathcal{R}\eta\beta} \lambda m, y.y\\ &=_{\mathcal{R}\eta\beta} \lambda m, y.(\mathcal{R}^{\sigma}_{\mathbf{N}}\,y\,\overline{f}_m)\,0\end{aligned}$$

$n + 1$

$$\begin{aligned}\Pi\,(n+1) &=_{\mathcal{R}\eta\beta} (\lambda n, p, m, y.p(m+1)(f\,m\,y))\,n\,(\Pi n)\\ &=_{\mathcal{R}\eta\beta} \lambda m, y.(\Pi n)\,(m+1)\,(f\,m\,y)\\ &=_{\mathcal{R}\eta\beta} \lambda m, y.(\lambda m, y.(\mathcal{R}^{\sigma}_{\mathbf{N}}\,y\,\overline{f}_m)n)\,(m+1)\,(f\,m\,y) \quad \text{by IH}\\ &=_{\mathcal{R}\eta\beta} \lambda m, y.(\mathcal{R}^{\sigma}_{\mathbf{N}}\,(f\,m\,y)\,\overline{f}_{m+1})\,n\\ &=_{\mathcal{R}\eta\beta} \lambda m, y.(\mathcal{R}^{\sigma}_{\mathbf{N}}\,(\overline{f}_m\,0\,y)\,\overline{f}_{m+1})\,n \qquad \text{by Def. 7.2.4}\\ &=_{\mathcal{R}\eta\beta} \lambda m, y.(\mathcal{R}^{\sigma}_{\mathbf{N}}\,y\,\overline{f}_m)(n+1) \qquad \text{by Prop. 7.2.5}\end{aligned}$$

□

Now, compared with previous step, we have to provide an *initial value* to $\Pi$ in order to get an equivalent program. According to the accumulator-based

approach, arguments $0, b$ roughly take the place of the continuation (function). See section 7.3 for more development on this remark.

**Corollary 7.2.7.** $\lambda n.\Pi\, n\, 0\, b =_{\mathcal{R}\eta\beta} (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$

Again, we still have to address the question, whether given a proof $F$ such that

$$[\![F]\!] = (\mathcal{R}^{\sigma}_{\mathbf{N}}\, b\, f)$$

it is possible to find $F'$ such that:

$$[\![F']\!] = \lambda n.(\Pi\, n\, 0\, b)?$$

Functions are very powerful tools, so it is not a surprise that going along without them has a cost. Actually, we can still achieve our goal, but the answer is now a little bit more elaborate.

Given two natural indexes $i$ , $j$, with $i + j = n$, $(\Pi\, i\, j)$ is a function that takes the witness for $A(j)$ and returns the witness for $A(i + j)$. So we expect $\Pi$ to be the computational content of a proof with end formula:

$$\forall n, m(A(m) \to A(n + m))$$

that use the proofs terms $M^{A(0)}$ and $N^{\forall n(A(n)\to A(n+1))}$ as assumptions. Let us prove this claim.

**Proposition 7.2.8.** $A(0) \to \forall n(A(n) \to A(n + 1)) \to \forall n, m(A(m) \to A(n + m))$

*Proof.* Assume $b : A(0)$ and $f : \forall n(A(n) \to A(n + 1))$. By induction on $n$:

$n = 0$ We have to prove

$$\forall m(A(m) \to A(m))$$

this is trivially proved by $(\lambda\, m, u.u)$.

$n + 1$ Let us assume $n$, the recursive call $p : \forall m(A(m) \to A(n + m))$, $m$ and the accumulator $y : A(m)$. We have to prove

$$A(n + m + 1)$$

Apply $f$ to $m$ and $y$ obtaining $(f\, m\, y) : A(m+1)$. Now apply $p$ to $(m+1)$ and $(f\, m\, y)$.

□

The accumulator-based program transformation provides us with a new proof of the induction principle over natural numbers:

**Proposition 7.2.9.** $A(0) \to \forall n(A(n) \to A(n+1)) \to \forall n A(n)$.

*Proof.* Assume $b : A(0)$, $f : \forall n(A(n) \to A(n+1))$ and $n$. To prove $A(n)$: instantiate the formula proved in Proposition 7.2.8 on $n$, 0 and $b : A(0)$ □

We are done: the program extracted from the previous proof named as Ind_ACC, is the following:

```
[b,f,n] (Rec nat => nat => sigma => sigma)
         ([m,y]y)
         ([n,p,m,y] p (m+1)(f m y)) n 0 b
```

## 7.3 From Higher Order to First Order Computation

In this section, we answer positively to the question of the existence for some formal connection between Ind_CONT and Ind_ACC. The link between the two of them relies on *Defunctionalization*. This program transformation, first introduced by Reynolds in the early 1970's [32] and later on extensively studied by Danvy [15], is a whole program transformation to turn higher-order into first-order functional programs, that is to transform programs where functions may be anonymous, given as arguments to other functions and returned as results, into programs where none of the functions involved accept arguments or produce results that are functions. Let us consider the following simple example taken from [15]:

```
(* aux : (nat -> nat) -> nat *)
let aux f = (f 1) + (f 10)

(* main : nat * nat * bool -> nat *)
let main x y b = aux (fun z -> x + z) *
                 aux (fun z -> if b then y + z else y * z)
```

The above function `aux` calls the higher order function $f$ twice: on 1 and 10 and returns the sum as its result. Also, the `main` function calls `aux` twice and returns the product of these calls. There are only two function abstractions and they occur in `main`.

Defunctionalizing this program amounts to defining a data type with two constructors, one for each function abstraction, and its associated apply function. The first function abstraction contains one free variable (`x`, of type `nat`), and therefore the first data-type constructor requires a natural. The second function abstraction contains two free variables (`y`, of type `nat`, and `b` of type `bool`), and therefore the second data-type constructor requires an integer and a boolean.

In main, the first abstraction is thus introduced with the first constructor and the value of x, and the second abstraction with the second constructor and the values of y and b.

To the functional argument used in aux, corresponds a pattern matching done by the following apply function:

```
type lam = LAM1 of nat | LAM2 of nat * bool

(* apply : lam * nat -> nat *)
let apply l z =
  match l with
  | LAM1 x -> x + z
  | LAM2 y b -> if b then y + z else y - z

(* aux_def : lam -> nat *)
let aux_def f = apply f 1 + apply f 10

(* main_def : nat * int * bool -> nat *)
let main_def x y b = aux_def (LAM1 x) * aux_def (LAM2 y b)
```

Now let us apply defunctionalization to Ind_CONT. We introduce the algebra path_nat (below) to represent the *initial* continuation $\lambda x.x$ and the *intermediate* continuation $\lambda u.k(f\,n\,u)$.

```
type path_nat = TOP | UP of path_nat * nat
```

Each constructor has as much parameters as free variables occurring in the corresponding continuation function. Finally the call (k b) in Ind_CONT is replaced by the *apply* function (here is anonymous) that dispatches over the path_nat constructors. We named the defunctionalization of Ind_CONT by Insd_Def_CONT and it is listed below:

```
[n](Rec nat => path_nat => sigma
     [q] (Rec path_nat => sigma => sigma
           [y] y
           [m,q',p,y] (p (f m y))) q b
     [n,p,q] (p (UP q n))) n TOP
```

Now the question is: from which proof is it possible to extract Ind_Def_CONT? Given $q$ of type path_nat and $y$ "of type" $A(n)$ the inner procedure would be expected to return an element of type $A(n)$ when $q =$ TOP and an element of type $A(n+m+1)$ when $q =$ (UP (...(UP TOP $n+m$)...) $n$). But $q$ does not depend explicitly on $n$, so given $y$ and $p$ alone one cannot guess anything about the type of the returned value. In order to state this link between the

above two inputs we need to quantify *non computationally* over an additional parameter as showed in the theorem below. In order to do that, let us before introduce the following notation.

**Definition 7.3.1.** *Given $p$ and $q$ of type* `path_nat` *the "degree" of $q$ with respect to $p$ is defined by the following partial function:*

$$\sharp_p(q) = \begin{cases} \sharp_p(p) = 0 \\ \sharp_p(\mathit{TOP}) = \mathit{Undef} & \mathit{if} p \neq \mathit{TOP} \\ \sharp_p((\mathit{UP}\, q\, n)) = 1 + \sharp(q) \end{cases}$$

**Definition 7.3.2.** *Given $x$ and $p$ of type* `path_nat` *and a natural $n$, we say that $x$ has a "good shape" with respect to $p$ at level $n$ when*

$$\mathit{GoodShape}(x,p,n) \iff \begin{cases} p = x \\ p \neq x = (\mathit{UP}\, q\, l) \wedge (l = n) \wedge \mathit{GoodShape}(q,p,n+1) \end{cases}$$

In the following we adopt the following notation: by $\mathcal{C}[t]$ we indicate a `path_nat` term that contain an occurrence of the term $t$. So for example if $\mathcal{C}[t]$=`(UP(UP TOP` $j$ `)`$i$`)`, for some naturals $i$ and $j$, then $t$ it could be `TOP`, `(UP TOP` $j$ `)` or $\mathcal{C}[t]$ it self.

**Theorem 7.3.1.** $A(0) \to \forall n(A(n) \to A(n{+}1)) \to \forall x \forall^{nc} n\, \mathit{GoodShape}(x, \mathit{TOP}, n) \to A(n) \to A(n + \sharp_{\mathit{TOP}}(x))$

*Proof.* By induction over $x$.

$x = \mathtt{TOP}$ Assume $n$, $u : A(n)$ and $\mathsf{GoodShape}(\mathtt{TOP}, \mathtt{TOP}, n)$. The thesis follows by $u$.

$x = (\mathtt{UP}\, q\, l)$ Assume $\mathsf{p} : \forall^{\mathsf{nc}} n.\, \mathsf{GoodShape}(q, \mathtt{TOP}, n) \to A(n) \to A(n + \sharp(q))$, $n$, $\mathsf{gs} : \mathsf{GoodShape}((\mathtt{UP}\, q\, l), \mathtt{TOP}, n)$ and $y : A(n)$. By $\mathsf{gs}$ and definition 7.3.1 follows $l = n$ and $\mathsf{gs}' : \mathsf{GoodShape}(q, \mathtt{TOP}, n+1)$. Instantiate $f$ on $l$ and $A(n)$ ($l$ is equal to $n$) obtaining $(f\, l\, y) : A(n+1)$. To prove the thesis, it remains to instantiate $\mathsf{p}$ on $n+1$, $\mathsf{gs}'$ and $(f\, l\, y)$.

□

The program extracted from theorem 7.3.1 is Ind_Def_CONT but we are not done yet: the theorem below shows as Ind_Def_CONT needs some additional simplification. In the following lines we will favor the presentation $(\lambda n.\mathcal{P}\, n\, \mathsf{TOP})$ in place of Ind_Def_CONT.

**Theorem 7.3.2.** *For all $n, p^{\mathtt{path\_nat}}, ACC^{\mathtt{path\_nat}}$, if*

$$(\lambda n.\mathcal{P}\, n\, p)\,(n+1) =_{\mathcal{R}\eta\beta} \mathcal{P}\, 0\, ACC$$

*then $\mathit{GoodShape}(ACC, p, 0)$ and $\sharp_p ACC = n + 1$.*

*Proof.* By induction on $n$.

$n = 0$ $(\lambda n.\mathcal{P}\, n\, p)\, 1$ rewrite to $\mathcal{P}\, 0\, (\mathtt{UP}\, p\, 0)$ in one step.

$n > 0$ Assume IH: $\forall p, \mathsf{ACC}$, if $(\lambda n.\mathcal{P}\, n\, p)\, (n+1) =_{\mathcal{R}\eta\beta} \mathcal{P}\, 0\, \mathsf{ACC}$ then $\sharp_p \mathsf{ACC} = n+1$ and $\mathsf{GoodShape}(\mathsf{ACC}, p, 0)$; assume $p$, ACC and ip:$(\lambda n.\mathcal{P}\, n\, \mathsf{p})\, (n+2) =_{\mathcal{R}\eta\beta} \mathcal{P}\, 0\, \mathsf{ACC}$. We have to prove $\mathsf{GoodShape}(\mathsf{ACC}, p, 0)$ and $\sharp_p \mathsf{ACC} = n+2$. It is just enough to see that $(\lambda n.\mathcal{P}\, n\, \mathsf{p})\, (n+2) =_{\mathcal{R}\eta\beta} (\lambda n.\mathcal{P}\, n\, (\mathtt{UP}\, p\, (n+1)))(n+1)$ and so by ip, we have $\mathsf{ip}' : (\lambda n.\mathcal{P}\, n\, (\mathtt{UP}\, p\, (n+1)))(n+1) =_{\mathcal{R}\eta\beta} \mathcal{P}\, 0\, \mathsf{ACC}$. Then instantiating IH on $(\mathtt{UP}\, p\, (n+1))$ and ACC, and by $\mathsf{ip}'$ we have that $\mathsf{GoodShape}(\mathsf{ACC}, (\mathtt{UP}\, p\, (n+1)), 0)$ and $\sharp_{(\mathtt{UP}\, p\, (n+1))}\mathsf{ACC} = n+1$. It follows that $\mathsf{ACC} = \mathcal{C}[(\mathtt{UP}\, (\mathtt{UP}\, p\, (n+1))\, n)]$, for some `path_nat` term $\mathcal{C}$, that is $\sharp_p(\mathsf{ACC}) = n+2$ and $\mathsf{GoodShape}(\mathsf{ACC}, p, 0)$.

□

As a corollary of theorem 7.3.2, we have that, for $p = \mathtt{TOP}$ the expression $(\lambda n.\mathcal{P}\, n\, \mathtt{TOP})(n+1)$, that is $\mathsf{Ind\_Def\_CONT}(n+1)$, rewrites to $(\mathcal{P}\, 0\, \mathsf{ACC})$ with $\mathsf{GoodShape}(\mathsf{ACC}, \mathtt{TOP}, 0)$ and $\sharp_{\mathtt{TOP}}(\mathsf{ACC}) = n+1$. A data structure like `type_nat` is too complex to store this particular simple data. So we replace `type_nat` by **N** in Ind_Def_CPS according to the informal correspondence:

$$\begin{array}{rcl} \mathsf{TOP} & \rightsquigarrow & 0 \\ (\mathtt{UP}\, \mathsf{TOP}\, n) & \rightsquigarrow & 1 \\ \vdots & & \vdots \\ (\mathtt{UP}(\ldots(\mathtt{UP}\, \mathsf{TOP}\, n)\ldots)0) & \rightsquigarrow & n+1 \end{array}$$

obtaining the code Ind_Intermediate_ACC, listed below:

```
[n] (Rec nat => nat => sigma
     [q] (Rec nat => nat => sigma => sigma
          [m,y] y
          [q',p,m,y] (p (m+1) (f m y))) q 0 b
     [n,p,q] (p (q+1))) n 0
```

This procedure still performs some redundant computations: the outer recursion runs over $n$, so the accumulator parameter $q$ ranges from 0 to $n$. At this point the inner routine (that will return the final result) is called on $q$, now equal to $n$. This is equivalent to calling directly the subroutine over $n$, which corresponds to Ind_ACC as expected.

## 7.4 Case Study

Let us consider now a more elaborated example taken from Bioinformatics. This is an area where the *correctness* and the *efficiency* of programs plays

a crucial role: *efficiency* because DNA sequences are really huge and getting lower complexity class is essential, *correctness* because we need to trust programs and we cannot check their results by hand. An important line of research is the "Sequence Analysis", which is concerned with locating biologically meaningful segments in DNA sequences. In this context, we will treat the so-called "Maximal Scoring Subsequence" (MSS) Problem. For a sequence of real numbers, we are looking for a contiguous sub-sequence such that the sum of its elements is maximal over all sub-sequences. Several authors have investigated that problem or a variation thereof, see, e.g., [16, 11, 18, 25, 42]

The MSS problem has various applications in Bioinformatics and we will mention only a few of them. The GC content in DNA of all organisms varies from 25% to 75%, where, e.g., genes are usually located in region with a high GC content. Such regions can easily be determined with a MSS algorithm, where the bases G and C get a positive, while the bases A and T get a negative value. Also in comparative genomics, the sequence similarity for corresponding exons between human and mouse is up to 85%, while for introns it is as low as 35%. Using the Smith-Waterman local alignment algorithm such regions with high similarity can be roughly determined, but a refinement in a post-processing step using variations of MSS algorithms are helpful to eliminate sub-regions with a low similarity. Furthermore, strongly conserved regions of a multiple sequence alignment can be found using MSS algorithms, where each column will be scored based on a suitable similarity measure. In transmembrane proteins, the more hydrophobic regions of the protein are usually located inside the membrane and more hydrophilic regions are located outside. Thus, locating hydrophobic regions using MSS algorithms are helpful for a first rough structure resolution of transmembrane proteins, where hydrophobic amino acids get a positive and hydrophilic a negative value. For a detailed list of applications in biomolecular sequence analysis, see [25], for example.

### 7.4.1 The MSS Problem

The MSS (Maximal Scoring Subsequence) problem, in its most general presentation, can be explained as follows:

**MSS Problem** : Given a list $l$ of real numbers, find an interval $(i, k)$ (with $i \leq k \leq |l| - 1$) such that

$$\sum_{j=i'}^{k'} l\,[j] \leq \sum_{j=i}^{k} l\,[j]$$

for every $(i', k')$ (with $i' \leq k' \leq |l| - 1$). The problem doesn't admit solutions for all the inputs, in fact on the empty list there is no solution.

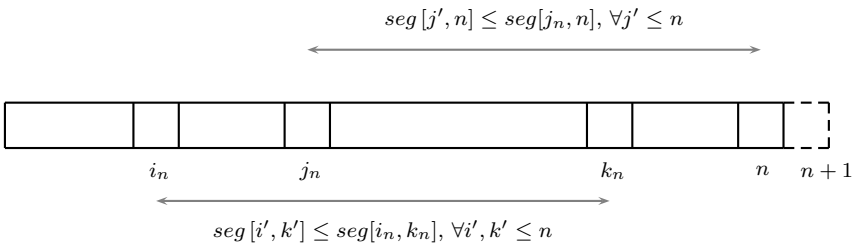



Figure 7.1: The witnesses $i_n$,$j_n$ and $k_n$ at step $n$ of the induction

Here we report on a variant of the MSS problem first proposed in [2, 35].

**MSS Problem Instance** :Given the function $seg : \mathbb{N}\times\mathbb{N} \to \mathbb{X}$ defined on $[0, \ldots, n]\times[0, \ldots, n]$, find the interval $(i, k)$,(with $i \leq k \leq n$) such that

$$seg\,[i', j'] \leq_{\mathbb{X}} seg\,[i, j]$$

for every $(i', k')$, (with $i' \leq k' \leq n$). This time the problem admits solution on each natural input $n$. Here $\mathbb{X}$ is a set on which we can define a total order relation $\leq_{\mathbb{X}}$. Moreover we require $seg$ to have the following property:

$$\mathsf{AX} = \forall n, i, j.\, seg[i, n] \leq_{\mathbb{X}} seg[j, n] \to seg[i, (\mathsf{Succ}\ n)] \leq_{\mathbb{X}} seg[j, (\mathsf{Succ}\ n)]$$

**Theorem 7.4.1.** *For all $n$*

$$\exists i, k((i \leq k \leq n) \wedge \forall i', k'((i' \leq k' \leq n) \to seg[i', k'] \leq_{\mathbb{X}} seg[i, k]) \quad (7.2)$$
$$\exists j((j \leq n) \wedge \forall j'((j' \leq n) \to (seg[j', n] \leq_{\mathbb{X}} seg[j, n])))) \quad (7.3)$$

*Proof.* By induction on $n$.

$\underline{n = 0}$ We set $i = k = j = 0$.

$\underline{n + 1}$ Assume (7.2) and (7.3) hold for $n$ (hypothesis $\mathsf{IH}^1_n$,$\mathsf{IH}^2_n$). Let $(i_n, k_n)$ and $j_n$ be the segment and the value that satisfy $\mathsf{IH}^1_n$ and $\mathsf{IH}^2_n$ respectively (see picture in Figure 7.1) By $\mathsf{IH}^2_n$, for an arbitrary $j' \leq n$

$$seg[j', n] \leq_{\mathbb{X}} seg[j_n, n] \quad (7.4)$$

Instantiating Ax on $n$,$j'$,$j_n$ and (7.4),

$$seg[j', n+1] \leq_{\mathbb{X}} seg[j_n, n+1]$$

The witness for $\mathsf{IH}^2_{n+1}$ is given by:

$$j_{n+1} = \begin{cases} j_n & seg[n+1, n+1] \leq_{\mathbb{X}} seg[j_n, n+1] \\ (n+1) & seg[n+1, n+1] \not\leq_{\mathbb{X}} seg[j_n, n+1] \end{cases}$$

We have to prove that $j_{n+1}$ satisfies,

$$\forall j'.(j' \leq (n+1)) \rightarrow seg[j', (n+1)] \leq_{\mathbb{X}} seg[j_{n+1}, (n+1)])$$

This has to be proved both for $j' \leq n$ and $j' = (n+1)$. Both cases follow straightforwardly from $\mathsf{IH}^2_n$ and the construction of $j_{n+1}$. The new maximal segment, is given by:

$$(i_{n+1}, j_{n+1}) = \begin{cases} (i_n, k_n) & seg[j_{n+1}, n+1] \leq_{\mathbb{X}} seg[i_n, k_n] \\ (j_{n+1}, n+1) & seg[j_{n+1}, n+1] \not\leq_{\mathbb{X}} seg[i_n, k_n] \end{cases}$$

Again, we have to prove that $(i_{n+1}, k_{n+1})$ satisfies,

$$\forall i', k'(i' \leq k' \leq (n+1)) \rightarrow seg[i', k'] \leq_{\mathbb{X}} seg[i_{n+1}, k_{n+1}]$$

This property has to be proved both for $(i' \leq k' \leq n)$ and $(i' \leq k' = n+1)$. Both cases follows from $\mathsf{IH}^1_n$, $\mathsf{IH}^2_n$, and the construction of $(i_{n+1}, k_{n+1})$

□

The program extracted from the previous proof, named MSS, is the following:

```
(Rec nat => sigma
   (0,0,0)
   [n,(i,j,k)]
      LET m = (if(seg[n+1, n+1] <= seg[j, n+1]) j (n+1))
      IN if(seg[m,n+1] <= seg[i,k]) (i,m,k) (m,m,n+1))
```

With `seg` some fixed function. The above algorithm makes use of the expression (`LET r IN s`). This is actually *syntactic sugar*: although it does not belong to our term language, Minlog allows the user to make use of it. This is irrelevant in the context of this section, and the reader is referred to [10] for a a further development on that issue.

By the following extension of the definition 7.2.1:

3′. *if* $t \equiv (\mathsf{LET}\ r\ \mathsf{IN}\ s)$ *then* $s$ *is a tail expression.*

and w.r.t. definition 3.3, the program MSS is not *tail* recursive.

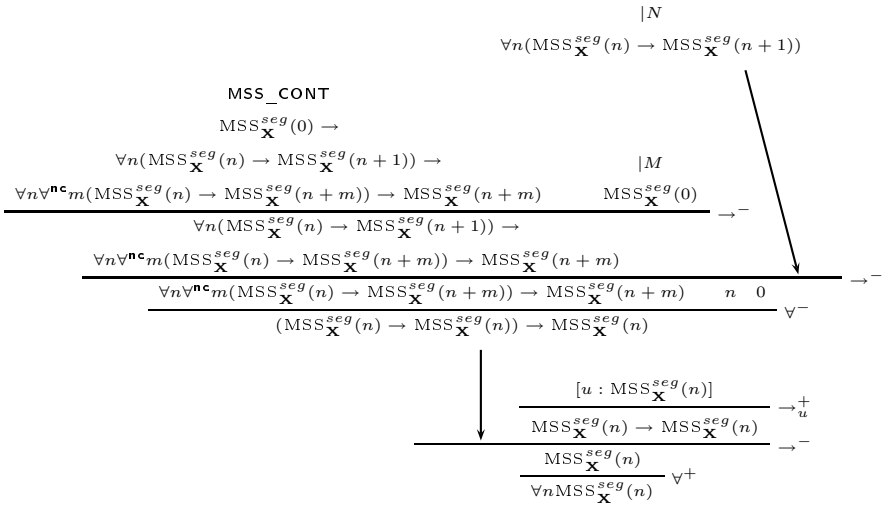
### 7.4.2 Generation of a Continuation/Accumulator Based MSS-Program

We apply the transformations proposed in section 7.2.1 and 7.2.2 to the proof of the theorem 7.4.1 in order to extract respectively a continuation and an accumulator based version of the MSS program. We first consider the extraction of a continuation based version of the MSS program. Before to do that, let's name the following formula,

$$\begin{aligned}&\forall n \exists i,k((i \leq k \leq n) \wedge \\ &\quad \forall i',k'((i' \leq k' \leq n) \rightarrow seg[i',k'] \leq_{\mathbb{X}} seg[i,k]) \wedge \\ &\quad\quad \exists j((j \leq n) \wedge \forall j'((j' \leq n) \rightarrow (seg[j',n] \leq_{\mathbb{X}} seg[j,n]))))\end{aligned}$$

with $\forall n \mathrm{MSS}_{\mathbf{X}}^{seg}(n)$. Moreover we name the base and the step of the inductive proof of theorem 7.4.1 respectively as $M$ and $N$. Clearly $M$ has type $\mathrm{MSS}_{\mathbf{X}}^{seg}(0)$ and $N$ has type $\forall n(\mathrm{MSS}_{\mathbf{X}}^{seg}(n) \rightarrow \mathrm{MSS}_{\mathbf{X}}^{seg}(n+1))$.

Now, let instantiate $A(n)$ in Proposition 7.2.3 with $\mathrm{MSS}_{\mathbf{X}}^{seg}(n)$. We name the proof of the Proposition 7.2.3 so istantiated as MSS_CONT. At this point, following the idea proposed in Proposition 7.2.4 we build the following proof-tree:




The program extracted from the above proof is the continuation based version of the MSS program:

```
([n]
  (Rec nat => (sigma => sigma) => sigma
```

```
    [k] k (0,0,0)
    [n,p,k] p ([(i,j,k)]
                LET m = if (seg[n+1, n+1] <= seg[j, n+1]) j (n+1)
                IN if (seg[m,n+1] <= seg[i,k]) (i,m,k) (m,m,n+1))))
n [x]x
```

For the extraction of an accumulator based version of the MSS program we follow the same idea. Before we instantiate $A(n)$ in Proposition 7.2.8 with $\mathrm{MSS}_{\mathbf{X}}^{seg}(n)$. We name the proof of the Proposition 7.2.8 so istantiated as MSS_ACC. Now adapting the proof of Proposition 7.2.9 we build the following proof-tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall n(\mathrm{MSS}_{\mathbf{X}}^{seg}(n) \to \mathrm{MSS}_{\mathbf{X}}^{seg}(n+1)) \to \forall n, m(\mathrm{MSS}_{\mathbf{X}}^{seg}(m) \to \mathrm{MSS}_{\mathbf{X}}^{seg}(n+m))}\ \mathrm{MSS\_CONT} \quad |N}{\forall n(\mathrm{MSS}_{\mathbf{X}}^{seg}(n) \to \mathrm{MSS}_{\mathbf{X}}^{seg}(n+1))} \to^{-} \quad n \quad 0}{\forall n, m(\mathrm{MSS}_{\mathbf{X}}^{seg}(m) \to \mathrm{MSS}_{\mathbf{X}}^{seg}(n+m))} \forall^{-}}{(\mathrm{MSS}_{\mathbf{X}}^{seg}(0) \to \mathrm{MSS}_{\mathbf{X}}^{seg}(n))} \quad |M \quad \mathrm{MSS}_{\mathbf{X}}^{seg}(0)}{\mathrm{MSS}_{\mathbf{X}}^{seg}(n)}}{\forall n \mathrm{MSS}_{\mathbf{X}}^{seg}(n)} \forall^{+}$$

The program extracted from the above proof is the accumulator based version of the MSS program

```
([n]
  (Rec nat => nat => sigma => sigma
   [m,y] y
   [n,p,m,(i,j,k)]
         p (m+1) LET m = (if (seg[n+1, n+1] <= seg[j, n+1]) j (n+1))
                 IN if (seg[m,n+1] <= seg[i,k]) (i,m,k) (m,m,n+1))))
 n 0 (0,0,0)
```

Both the continuation and accumulator version of the MSS program are tail recursive, as the result of automatic transformation from the proof of the theorem 7.4.1. This way, we have ensured these are still correct implementations of the abstract algorithm while being more efficient in the same time.

# 8 Beyond Primitive Recursion

## 8.1 Introduction

In this chapter[1] we extend what we have seen in the previous chapter. Following the pioneering work of Manna and Waldinger's [27] we introduce several induction principles over natural numbers and we will investigate how it is possible to express each one in terms of the others, both from a programming and a proof-theoretic point of view. This represents a contribution with respect to [27]. Moreover we will show how it is possible to turn each induction principle into an equivalent one, but from which it is possible to automatically synthesize a tail recursive program.

For readability reasons part of the code presented in this section will be written with the ML syntax.

### 8.1.1 Up Primitive Recursive Induction

Here is the proof principle for primitive recursion:

$$\frac{\begin{array}{cc} |Z & |S \\ P(0) & \forall n(P(n) \rightarrow P(n+1)) \end{array}}{\forall n P(n)} \text{ (up-prim-rec)}$$

Manna and Waldinger refer to it as 'going up' since $P(n)$ is needed to deduce $P(n+1)$. The corresponding synthesized functional `Up.prim_rec` is displayed in Figure 8.1. There, `z` is extracted from $[\![Z]\!]$ and `s` from $[\![S]\!]$. The computation is driven by the input variable `n`: computing the result for $n$ requires the result for $n-1$ to be computed, until the base case $n = 0$ is reached in a trail of nested applications of the function denoted by `s`.

The recursive definition of the factorial function is a straightforward example of primitive recursion, and is obtained as an instance of `Up.prim_rec` where `z` is instantiated with identity element for multiplication (`z = 1`) and `s` with the (curried) multiplication function (`s = fn i => fn c => (i + 1) * c`):

[1] The material in this chapter was developed in collaboration with Olivier Danvy during January 2009, during a visit to the Århus's Computer Science Department.

```
structure Up
= struct
    fun prim_rec n   (*: nat-> 'a *)
        = let fun visit m
                  = if m =0 then z else s (m - 1)(visit(m - 1))
          in visit n
          end

    fun prim_iter n  (* : nat ->'a *)
        = let fun visit m
                  = if m = 0 then z else s (visit (m - 1))
          in visit n
          end
  end
```

Figure 8.1: Synthesized up-induction functionals

```
fun up_prim_rec_fact n
    = let fun visit m
              = if m = 0 then 1 else m * (visit (m - 1))
      in visit n
      end
```

### 8.1.2 Up Primitive Iterative Induction

Here is the proof principle for primitive iteration:

$$\frac{\begin{array}{cc} |Z & |S \\ P(0) & \forall^{\mathsf{nc}} n(P(n) \to P(n+1)) \end{array}}{\forall n P(n)}\ (\text{up-prim-iter})$$

The difference between primitive and iterative iteration is that in the iterative case, we quantify non computationally over $n$ in the inductive step. One can then synthesize the functional for up primitive iteration `Up.prim_rec` in Figure 8.1. Again, there, `z` is extracted from $[\![Z]\!]$ and `s` from $[\![S]\!]$.

To define the factorial function as an instance of `Up.prim_iter` we must generalize Kleene's trick to compute the predecessor function over Church numerals. So instantiating `z` = `(1,1)` and `s` = `fn(i, c) =>(i+1,i*c)` in `Up.prim_iter` we obtain:

```
fun up_prim_iter_fact n
    = let fun visit m
```

```
    = if m = 0 then (1,1) else let val (i,c)=visit(m - 1)
                                  in (i + 1,i *c)
                                 end
in #2 (visit n)
end
```

### 8.1.3 Down Primitive Recursive Induction

Manna and Waldlinger also present a 'going down' version of primitive recursion:

$$\frac{\begin{array}{cc} |Z & |S \\ Q(n) & \forall m(Q(m+1) \to Q(m)) \end{array}}{Q(0)}\ (\text{down-prim-rec})$$

where $n$ *could* be a free variable in $Q$. They refer to it as 'going down' since $Q(n+1)$ is needed to deduce $Q(n)$.

The idea is that the property $\forall n P(n)$ is proved using a predicate $Q(m)$ such that $Q(0)$ reduces to $P(n)$ (noted $Q(0) \rightsquigarrow P(n)$). This induction principle is then applied to $Q(0)$. The challenging point here is that a kind of eureka step is required in order to find a satisfactory predicate $Q$.

So, given the proof of $Q(0)$ in terms of $M^{\exists m Q(n)}$ and $N^{\forall m(Q(m+1)\to Q(m))}$, we prove $\forall n P(n)$ by

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}|R\\ P(n)\end{array}}{Q(0)\to P(n)}\to^{+} \qquad \begin{array}{c}\vdots\\ Q(0)\end{array}}{P(n)}\to^{-}}{\forall n P(n)}\forall^{+}$$

Here we require the normalization of the code extracted from the proof-term $\lambda u^{Q(0)} R^{P(n)}$ to be equal to the identity function. This is because we assume $Q(z)$ to be a predicate that, when instantiated with 0, can be rewritten into $P(n)$ in a finite number of steps, using an opportune set of rewriting rules. This process of simplification is performed using the following, and only the following axiom:

$$\text{Eq-Compat} : \forall x_1, x_2 (x_1 \rightsquigarrow x_2 \to P(x_1) \to P(x_2))$$

where $\rightsquigarrow$ denotes a binary relation and $P$ a generic predicate symbol. This axiom says that, if we know that a given term (bounded by $x_1$) is in relation with another term (bounded by $x_2$) – for example the equality relation – and

```
structure Down
= struct
    fun prim_rec n  (* : nat -> 'a  *)
        = let fun visit m
                  = if m = n then z else s m (visit (m + 1))
          in visit 0
          end

    fun prim_iter n  (* : nat ->'a  *)
        = let fun visit m
                  = if m = n then z else s (visit (m + 1))
          in visit 0
          end
  end
```

Figure 8.2: Synthesized down-induction functionals

we know that $P(x_1)$ holds then we can conclude that $P(x_2)$ holds. Letting the computational content of the Eq-compat axiom be the identity function, it is clear that the program extracted from nested applications of Eq-compact, once normalized, will correspond to the identity function. Since the derivation above is a *detour*, we rewrite it in the following way:

$$\dfrac{\begin{array}{c}\vdots\\ Q(0)\\ |R\\ P(n)\end{array}}{\forall n P(n)}\,\forall^+$$

which can be read as the replacement of each open assumption $u^{Q(0)}$ in $R$ by the proof of $Q(0)$. The program extracted from the complete proof of $\forall n P(n)$ is the functional `Down.prim_rec` in Figure 8.2, where `z` *could* depend on `n` (hence the order of the parameters).

We now return to the factorial function over natural numbers:

$$\mathsf{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \mathsf{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Let us prove that $\forall n \exists m(m = \mathsf{fact}(n))$ by going-down primitive recursion. We assume $n$. In order to prove $\exists m(m = \mathsf{fact}(n))$, we design the new goal $\exists m(\mathsf{fact}(0) \times m = \mathsf{fact}(n))$. Applying the going-down primitive recursive induction principle to this formula requires us to prove the following two subgoals:

- $\exists m(\mathsf{fact}(n) \times m = \mathsf{fact}(n))$: It is sufficient to set $m = 1$.
- Now assume $y$ and ih : $\exists m(\mathsf{fact}(y+1) \times m = \mathsf{fact}(n))$. We prove $\exists m(\mathsf{fact}(y) \times m = \mathsf{fact}(n))$. By ih we know that there does exist an $m'$ such that $\mathsf{fact}(y+1) \times m' = \mathsf{fact}(n)$. Considering that $\mathsf{fact}(y+1) = (y+1) \times \mathsf{fact}(y)$, the thesis is proved for $m = (y+1) \times m'$.

The program extracted from this proof reads as follows:

```
fun down_prim_rec_fact n
    = let fun visit m
              = if m = n then 1 else (m + 1) * (visit (m + 1))
      in visit 0
      end
```

Correspondingly, this residual program is also obtained by specializing `Down.prim_rec` on `z` equal to the identity element for multiplication and `s` the (curried) multiplication function:

### 8.1.4 Down Primitive Iterative Induction

Here is the proof principle for primitive iteration:

$$\frac{\begin{array}{cc} |Z & |S \\ Q(n) & \forall^{\mathsf{nc}} m(Q(m+1) \to Q(m)) \end{array}}{Q(0)}\ (\text{down-prim-iter})$$

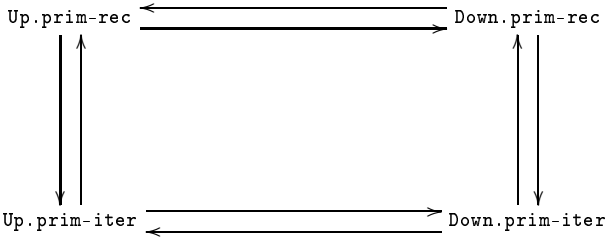
Again, the difference between primitive and iterative iteration is that in the iterative case, we quantify non computationally over $m$ in the inductive step. One can then synthesize the functional for down primitive iteration in Figure 8.2, where `n`, in the local definition of `visit`, is free.

Again, to define the factorial function as an instance of `Down.prim_iter` we use Goldberg and Reynolds's generalization of Kleene's trick to compute the predecessor function over Church numerals. So instantiating `z` = `(1,1)` and `s` = `fn(i,c) => (i+1, i*c)` in `Down.prim_iter` we obtain:

```
fun down_prim_iter_fact n
    = let fun visit m
          = if m=n then (1,1) else let val (i,c) = visit (m+1)
                                   in (i + 1, i * c)
                                   end
      in #2 (visit 0)
      end
```

## 8.2 Expressive Power

In this section we show that the induction principles reviewed in Section 8.1 share the same expressive power.




### 8.2.1 Up Primitive Iteration in Terms of Up Primitive Recursion

To simulate up primitive iteration in terms of up primitive recursion we instantiate the base and step of `Up.prim_rec` respectively by `z'` and `fn n=>fn y=>s'y` with `z'` and `s'` base and step of `Up.prim_item`:

```
fun up_prim_iter n
  = let fun visit m
          = if m =0 then z' else s'(visit(m - 1))
    in visit n
  end
```

Proof interpretation:

**Proposition 8.2.1.** *Given the proof*

$$\frac{\begin{array}{cc} |M & |N \\ P(0) & \forall^{nc} n(P(n) \to P(n+1)) \end{array}}{\forall n P(n)}\ (\textit{up-prim-iter})$$

*then there exists $M'$, $N'$ such that:*

$$\frac{\begin{array}{cc} |M' & |N' \\ P(0) & \forall n(P(n) \to P(n+1)) \end{array}}{\forall n P(n)}\ (\textit{up-prim-rec})$$

*with computational content equal to* `up_prim_iter`.

*Proof.*

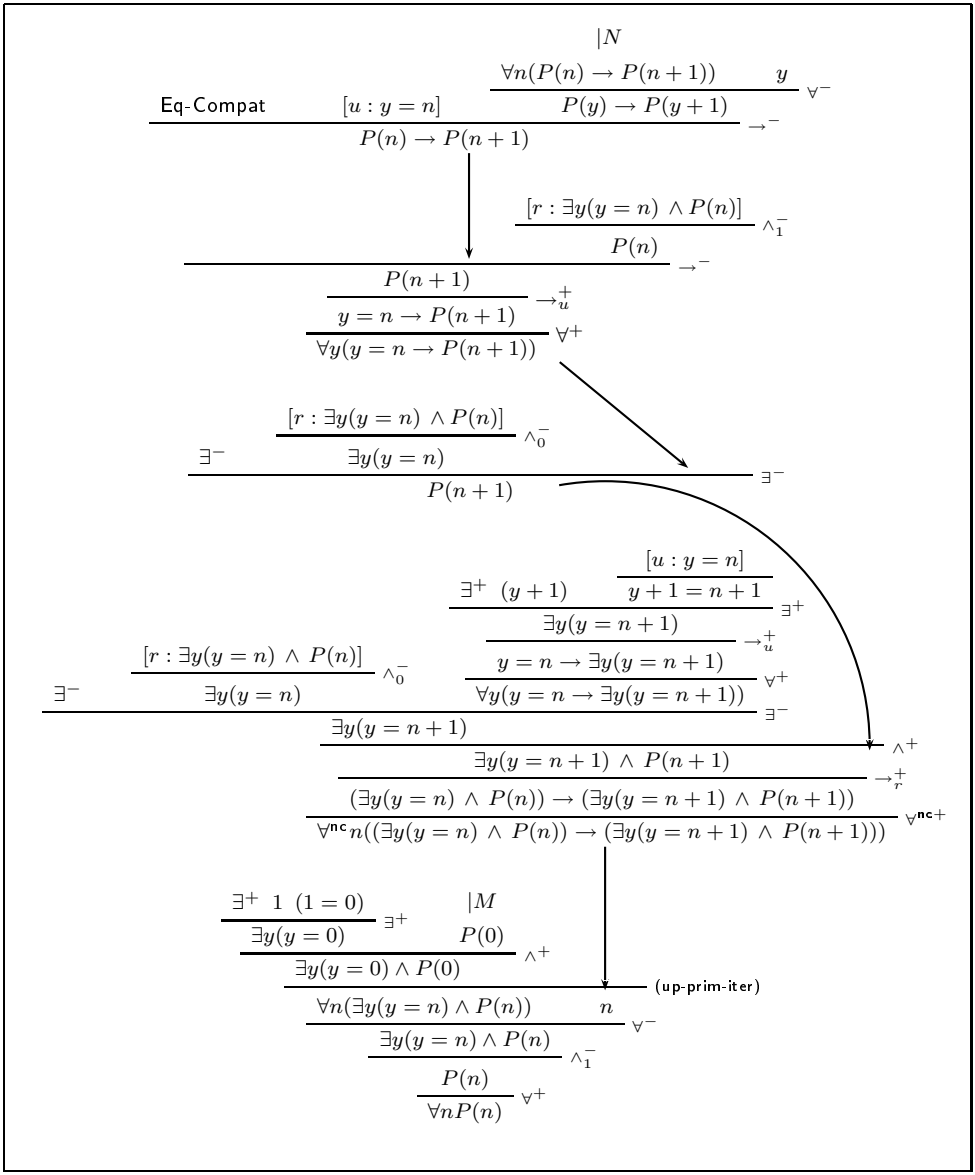



Figure 8.3: Simulation of up-prim-rec in term of up-prim-iter. The variable $n$ does not occur in content of the proof of the formula $(\exists y(y = n) \wedge P(n)) \rightarrow (\exists y(y = n + 1) \wedge P(n + 1))$, thus the $(\forall^{\mathsf{nc}+})$ inference results correct w.r.t. the definition given in section 2.1.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{|N}{\forall^{\mathsf{nc}} n(P(n) \to P(n+1))} \quad n}{P(n) \to P(n+1)} \forall^- \quad [u : P(n)]}{P(n+1)} \to^-}{P(n) \to P(n+1)} \to^+_u}{\dfrac{|M \qquad P(0)}{\forall n P(n)}}{\dfrac{\forall n(P(n) \to P(n+1))}{\forall n P(n)}} \forall^+ \quad (\text{up-prim-rec})$$

□

### 8.2.2 Up primitive Recursion in Terms of Up Primitive Iteration

To simulate up primitive recursion in terms of up primitive iteration we use Kleene's trick: we instantiate the base and step of `Up.prim_iter` respectively by `(0,z')` and `fn (j, c)=>(j + 1, s' j c)`, with `z'` and `s'` base and step of `Up.prim_rec`:

```
fun up_prim_rec n
  = let fun visit m
          = if m = 0 then (0,z')
            else let val (j,c) = (visit (m - 1))
                 in (j+1, s' j c) end
    in #2 (visit n)
    end)
```

Proof interpretation:

**Proposition 8.2.2.** *Given the proof*

$$\dfrac{\begin{array}{cc}|M & |N \\ P(0) & \forall n(P(n) \to P(n+1))\end{array}}{\forall n P(n)} \ (\text{up-prim-rec})$$

*then there exists $M'$, $N'$, $R$ such that:*

$$\dfrac{\begin{array}{cc}|M' & |N' \\ \exists y(y = 0) \wedge P(0) & \forall^{\mathsf{nc}} n(\exists y(y = n) \wedge P(n) \to \exists y(y = n+1) \wedge P(n+1))\end{array}}{\begin{array}{c}\forall n(\exists y(y = n) \wedge P(n)) \\ |R \\ \forall n P(n)\end{array}} \ (\text{up-prim-iter})$$

*and from which it is possible to extract* `up-prim-rec`.

*Proof.* See Figure 8.3. □

### 8.2.3 Up Primitive Recursion in Terms of Down Primitive Recursion

To simulate up primitive recursion in terms of down primitive recursion, we use Kleene's trick: we instantiate the base and step of `Down.prim_rec` respectively by `(0,z')` and `fn m =>fn(j,c)=>(j+1,s'jc)`, with `z'` and `s'` base and step of `Up.prim_rec`:

```
fun up_prim_rec' n
    = let fun visit m
          = if m = n then (0,z') else let val (j,c) = (visit (m + 1))
                                      in (j+1,s'jc) end
      in #2(visit 0)
      end
```

Proof interpretation:

**Proposition 8.2.3.** *Given the proof*

$$\dfrac{\begin{array}{cc} |M & |N \\ P(0) & \forall n(P(n) \to P(n+1)) \end{array}}{\forall n P(n)}\ \textit{(up-prim-rec)}$$

*then there exists $M'$, $N'$ such that:*

$$\dfrac{\dfrac{\dfrac{\begin{array}{cc} |M' & |N' \\ \exists z(z = n - n) \wedge P(n-n) & \forall y((\exists z(z = n-(y+1)) \wedge P(n-(y+1))) \to \\ & (\exists z(z = n - y) \wedge P(n-y))) \end{array}}{\exists z(z = n - 0) \wedge P(n-0)}\ \textit{(down-prim-rec)}}{P(n)}\ \wedge_1^-}{\forall n P(n)}\ \forall^+$$

*and from which it is possible to extract the procedure* `up-prim-rec'`.

*Proof.* See Figure 8.4. □

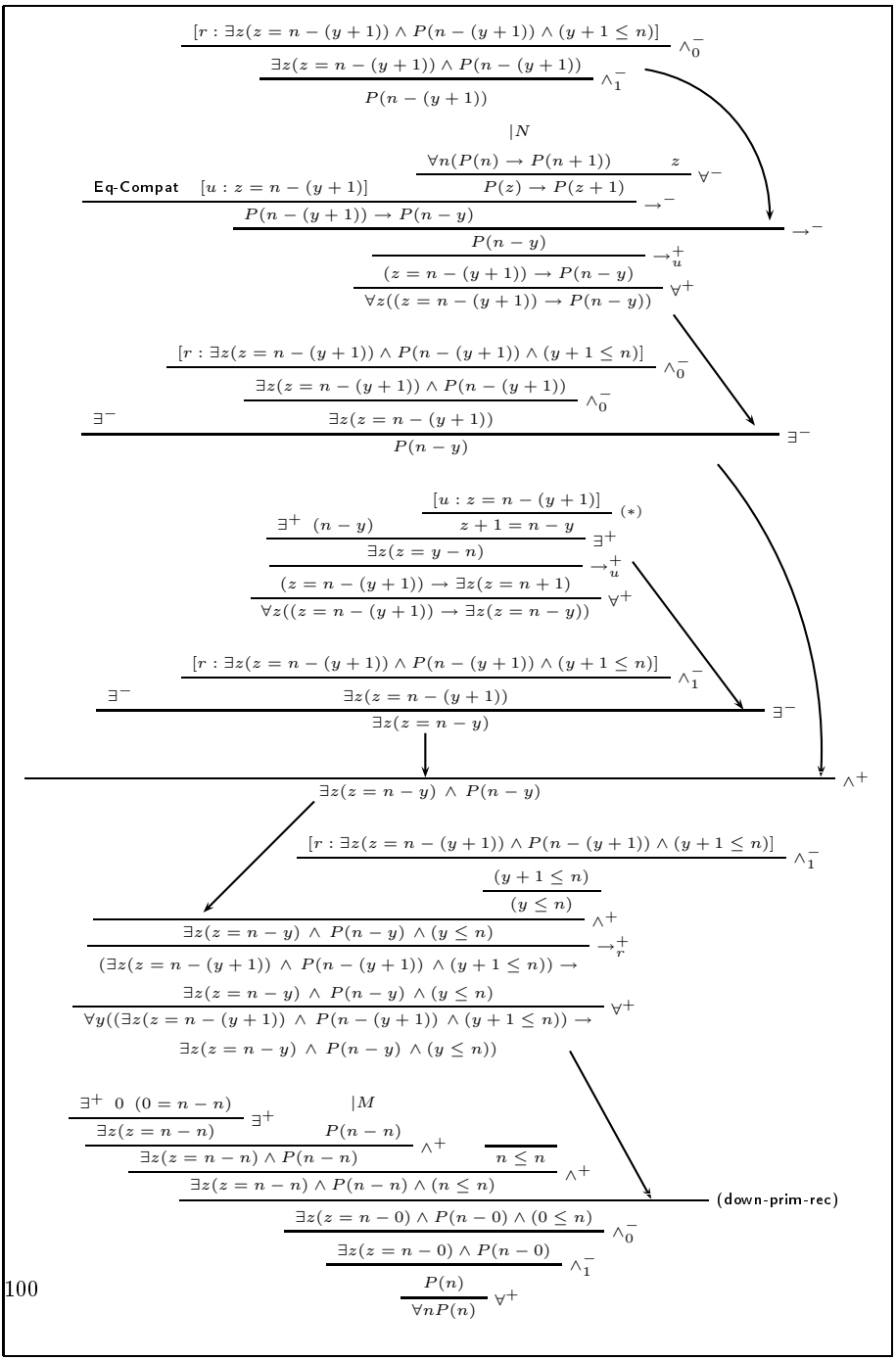





Figure 8.4:

### 8.2.4 Down Primitive Recursion in Terms of Up Primitive Recursion

To simulate down primitive recursion in terms of up primitive recursion, we instantiate the base and step of `Up.prim_rec` respectively by `(n,z')` (for some input parameter `n`) and `fn m=>fn (j,c)=> (j-1, s'(j-1)c)`, with `z'` and `s'` base and step of `Up.prim_rec`:

```
fun down_prim_rec n
  = let fun visit m
            = if m =0 then (z',n) else let val (j,c) = (visit(m - 1))
                                       in (j-1, s'(j-1)c) end
    in #2(visit n)
    end
```

Proof interpretation:

**Proposition 8.2.4.** *Given*

$$\dfrac{\dfrac{\begin{array}{cc}|M & |N\\ Q(n) & \forall m(Q(m+1)\to Q(m))\end{array}}{Q(0)}\,(\textit{down-prim-rec})\quad |R}{\dfrac{P(n)}{\forall n P(n)}\,\forall^+}$$

*we want to find the opportune $M'$, $N'$ such that if the proof of $Q(0)$ is substituted by*

$$\dfrac{\dfrac{\dfrac{\begin{array}{cc}|M' & |N'\\ \exists z(z=n)\wedge Q(n) & \forall y(((\exists z(z=n-y)\wedge Q(n-y))\to\\ & (\exists z(z=n-(y+1))\wedge Q(n-(y+1))))\end{array}}{\forall y(\exists z(z=n-y)\wedge Q(n-y))}\,(\textit{up-prim-rec})\quad n}{\exists z(z=0)\wedge Q(0)}\,\forall^-}{Q(0)}\,\wedge_1^-$$

*then the computational content of the resulting proof corresponds to* `down_prim_rec`.

*Proof.* We propose only a sketch because the structure of the proof is the same as the one displayed in Fig. 8.4. The idea is to prove the lemma

$$\forall y(\exists z(z=n-y)\wedge Q(n-y))$$

by up primitive recursion:

[Base $y = 0$] We have to prove $\exists z(z = n) \wedge Q(n)$. The left conjunct is proved by just introducing $n$ for $z$. The right conjunct is given my $M$.

[Step $y+1$] Let us assume $y$ and $z'$ such that $z' = n-y$ and $Q(n-y)$. We have to prove $\exists z(z = n-(y+1)) \wedge Q(n-(y+1))$. The left conjunct is proved introducing $z'-1$ for $z$. The right conjunct is proved by instantiating $N$ on $z'-1$, from which we deduce $Q(z') \to Q(z'-1)$ that can be rewritten as $Q(n-y) \to Q(n-y-1)$ by the induction hypothesis $z' = n-y$ and finally instantiating this formula on $Q(n-y)$.

□

### 8.2.5 Down Primitive Iteration in Terms of Down Primitive Recursion

To simulate down primitive iteration in terms of down primitive recursion, we instantiate the base and step of `Down.prim_rec` respectively by `z'` and `fn j =>fn c=>s'c`, with `z'` and `s'` base and step of `Down.prim_iter`:

```
fun down_prim_iter n
 = let fun visit m
                 = if m = n then z' else s'(visit (m + 1))
       in visit 0
       end
```

Proof interpretation:
*Proposition* 8.2.1. Given

$$\dfrac{\dfrac{\begin{array}{cc}|M & |N\\ Q(n) & \forall^{\mathsf{nc}} m(Q(m+1) \to Q(m))\end{array}}{Q(0)}\ \text{(down-prim-iter)}}{\dfrac{|R}{\dfrac{P(n)}{\forall n P(n)}\ \forall^+}}$$

we want to find the opportune $M'$, $N'$ such that if the proof of $Q(0)$ is replacled by

$$\dfrac{\begin{array}{cc}|M' & |N'\\ Q(n) & \forall m(Q(m+1) \to Q(m))\end{array}}{Q(0)}\ \text{(down-prim-rec)}$$

then computation content of the transformed proof is equal to `down_prim_iter`.

*Proof.* The structure of the proof is similar to that of Prop. 8.2.1. We simply set $M'$ equal to $M$ and $N'$ equal to the proof term $\lambda m, u^{Q(m+1)}(N^{\forall^{\mathsf{nc}} m(Q(m+1)\to Q(m))}\, m\, u)$. □

### 8.2.6 Down Primitive Recursion in Terms of Down Primitive Iteration

To simulate down primitive recursion in terms of down primitive iteration, we instantiate the base and step of `Down.prim_iter` respectively by `(n,z')` (for some given `n`) and `fn(j, c)=>(j -1, s(j-1)c)`, with `z'` and `s'` base and step of `Down.prim_rec`:

```
fun down_prim_rec' n
 = let fun visit m
             = if m = n then (n,z') else let val (j,c)=(visit (m+1))
                                          in (j -1, s(j-1)c)) end
   in #2(visit 0)
   end



    = #2 (Down.prim_iter n ((n, z), fn(j, c)=>(j -1, s(j-1)c)))
```

Proof interpretation:

**Proposition 8.2.5.** *Given the proof*

$$\dfrac{\dfrac{\begin{array}{cc}|M & |N\\ Q(n) & \forall m(Q(m+1)\to Q(m))\end{array}}{Q(0)}\ \textit{(down-prim-rec)}\quad |R}{\dfrac{P(n)}{\forall n P(n)}\ \forall^+}$$

*find $M'$, $N'$ and an appropriate $Q'$ such that*

$$\dfrac{\dfrac{\begin{array}{cc}|M' & |N'\\ Q'(n) & \forall^{\mathsf{nc}} m(Q'(m+1)\to Q'(m))\end{array}}{Q'(0)}\ \textit{(down-prim-iter)}\quad |R}{\dfrac{P(n)}{\forall n P(n)}\ \forall^+}$$

*and from which it is possible to extract* `down_prim_rec'`.

*Proof.* We propose only a sketch because the structure of the proof is the same as the one displayed in Fig. 8.3. The idea is to set

$$Q'(0) \equiv \exists y(y = 0) \wedge Q(0)$$

and prove $Q'(0)$ by up primitive iterative induction:

[Case $n$] We have to prove $\exists y(y = n) \wedge Q(n)$, which follows directly by $n = n$ and $M^{Q(n)}$.

[Case $m + 1 \to m$] Assume $m$ (which we quantify non computationally) and $y'$ such that $y' = m+1$ and $Q(m+1)$. We prove $\exists y(y = m) \wedge Q(m)$. For the left conjunct, it is enough to introduce $y' - 1$ for $y$. For the right conjunct, we need to instantiate $N$ with $(y' - 1)$, obtaining $Q(y') \to Q(y' - 1)$. By the assumption $y' = m + 1$, we have $Q(m + 1) \to Q(m)$ and instantiating it with $Q(m + 1)$ we obtain the thesis.

□

### 8.2.7 Up Primitive Iteration in Terms of Down Primitive Iteration

To simulate up primitive iteration in terms of down primitive iteration, we instantiate the base and step of `Down.prim_iter` respectively by `(0,z')` and `fn(j,c) => (j+1, s'c)`, with `z'` and `s'` base and step of `Up.prim_iter`:

```
fun up_prim_iter' n
      = let fun visit m
            = if m = n then (0,z') else let val (j,c) = (visit (m+1))
                                        in (j+1, s'c) end
        in #2(visit 0)
        end
```

This case is treated as the one in Section 8.2.4.

### 8.2.8 Down Primitive Iteration in Terms up Primitive Iteration

To simulate down primitive iteration in terms of up primitive iteration, we use Kleene's trick: we instantiate the base and step of `Up.prim_iter` respectively by `(n,z')` and `fn(j,c) => (j - 1, s' c)`, with `z'` and `s'` base and step of `Down.prim_iter`:

```
fun down_prim_iter' n (z, s)
= let fun visit m
          = if m = 0 then (n,z') else let val (j,c)=(visit(m -1))
                                      in (j-1, s'c) end
  in #2(visit n)
  end
```

This case is treated as the one in Section 8.2.3.

### 8.2.9 Summary and conclusion

We have shown proof theoretically how the original up versions and Manna and Waldinger's down versions of primitive recursion and primitive iteration are equivalent.

## 8.3 Primitive Recursion and Iteration with Accumulators

Here we present the proof-theoretical analogous of fold-left from functional programming with lists, where the result is accumulated at call time instead of at return time. We consider in turn the accumulator-based versions of each of the induction principles reviewed in Section 8.1.

### 8.3.1 Up Primitive Recursion with Accumulator

Here the problem is how to transform the following up primitive recursive induction principle,

$$\frac{\begin{array}{cc} |M & |N \\ P(0) & \forall n(P(n) \to P(n+1)) \end{array}}{\forall n P(n)}\ \text{(up-prim-rec)}$$

into another proof (of the same formula $\forall n P(n)$) but with a computational content that is the accumulator-based version of up primitive recursion:

```
fun up_prim_rec_acc n
    = let fun visit m j a
              = if m = 0 then a else visit (m - 1) (j + 1) (s j a)
      in visit n 0 z
      end
```

with `z` and `s` base and step of `Up.prim_rec`. In these definitions, we use and manipulate two accumulators: $j$, to count from 0 to $n$ and $a$, to store the partial result at step $j$. Obviously, for $j = n$ we have $a = s\,(n-1)(\ldots(s\,0\,z)\ \ldots)$.

So given a proof of $\forall n P(n)$ by the up primitive recursive induction principle in terms of $z : M^{P(0)}$ and $s : N^{\forall n(P(n) \to P(n+1))}$ we can build a new proof of $\forall n P(n)$ with content `up_prim_rec_acc` through the following two steps:

1. We prove the lemma $\forall n \forall m(P(m) \to P(n+m))$ by up primitive recursive induction:

   **Case** $n = 0$ We have to prove

   $$\forall m(P(m) \to P(m))$$

   which is trivially proved by $(\lambda m, u.u)$.

   **Case** $n + 1$ Let us assume $n$, the recursive call $p : \forall m(P(m) \to P(n+m))$, $m$ and the accumulator $y : P(m)$. We have to prove

   $$P(n + m + 1)$$

   Apply $s$ to $m$ and $y$, obtaining $(s\, m\, y) : P(m + 1)$. Now apply $p$ to $(m + 1)$ and $s\, m\, y$.

2. Finally we derive the initial formula $\forall n P(n)$ by assuming $n$ and instantiating the formula proved in the first step on $n$, 0 and $z : M^{P(0)}$.

### 8.3.2 Up Primitive Iteration with Accumulator

We follow the same schema as in Section 8.3.1. The only difference is that in the intermediate lemma (point 1), we have to quantify non computationally over $m$. In other words, we have to prove the modified intermediate lemma:

$$\forall n \forall^{\mathsf{nc}} m(P(m) \to P(n + m))$$

The synthesized program will embody the up primitive iterative induction principle with accumulator:

```
fun up_prim_iter_acc  n
    = let fun visit m a
              = if m = 0 then a else visit (m - 1) (s a)
      in visit n z
      end
```

with `z` and `s` base and step of `Up.prim_iter`.

### 8.3.3 Down Primitive Recursion with Accumulator

Here the problem is how to transform the following down primitive recursive induction principle,

$$\frac{|M \qquad |N}{Q(n) \qquad \forall y(Q(y+1) \to Q(y))}\ (\text{down-prim-rec})$$
$$Q(0)$$

into another proof, still of the formula $Q(0)$, but with a computational content that is the accumulator-based version of down primitive recursion:

```
fun down_prim_rec_acc n
    = let fun visit m j a
                  = if m = n then a else visit (m+1)(j-1)(s(j-1)a)
      in visit 0 n z
      end
```

with `z` and `s` base and step of `Down.prim_rec`. We propose here an approach similar to the one in Section 8.3.1. The function `down_prim_rec_acc` is equipped with two additional accumulators, indicated with the letters `j` and `a`. The first one is initialized with $n$ at the beginning of the computation and decreased of 1 in each iteration, and the second accumulator, initialized with $z$, of type $P(n)$, is dedicated to store the partial results. The proof from which it is possible to synthesize `up_prim_rec_acc` is based on the following two steps:

1. We prove the intermediate lemma $\forall i(Q(i) \to Q((i+0)-n))$ by down primitive recursive induction:

   **Case** $y = n$ We have to prove $\forall i(Q(i) \to Q(i))$ that is given by construction by the following proof term $\lambda i, u^{Q(i)} u$

   **Case** $y + 1 \to y$ Given $y$, the induction hypothesis visit : $\forall i(Q(i) \to Q((i+y+1)-n))$, $i$ and $\mathtt{u} : Q(i)$, we prove $Q((i+y)-n)$ by constructing the following proof term: $(\text{visit}\ (i-1)\ (N^{\forall y(Q(y+1)\to Q(y))}\ (i-1)\ u))^{Q((i+y)-n)}$.

2. We instantiate the proof of the formula $\forall i(Q(i) \to Q((i+0)-n))$ on $n$ and on $z^{Q(n)}$, obtaining $Q(0)$.

### 8.3.4 Down Primitive Iteration with Accumulator

We follow the same schema as in section8.3.3. The only difference is that in the intermediate lemma (point 1), we have to quantify non computationally over $i$. In other words, we have to prove the modified intermediate lemma:

$$\forall^{\mathsf{nc}} i(Q(i) \to P((i+0)-n))$$

The procedure extracted from this new proof is the following down primitive iteration principle with accumulator:

```
fun down_prim_iter_acc' n
    = let fun visit m a
              = if m = n then a else visit (m + 1) (s a)
      in visit 0 z
      end
```

with `z` and `s` base and step of `Down.prim_iter`.

### 8.3.5 Summary and Conclusion

We have presented the accumulator-based versions of Manna and Waldinger's going-up and going-down primitive recursion and primitive iteration reviewed in Section 8.1.

## 8.4 Case Study: The Factorial Function

In this section we put into practice what we have seen so far on a case study. We prove by up primitive induction over natural numbers that $\forall n \exists y(y = \mathsf{Fact}\,(n))$ (definition of Fact in 1.2):

$$\dfrac{\dfrac{}{\exists^{+}\quad 1 = \mathsf{Fact}(0)}\;\exists^{+}}{\exists y(y = \mathsf{Fact}(0))} \qquad \dfrac{\dfrac{\exists^{-}\quad [v : \exists y(y = \mathsf{Fact}(n))] \qquad \dfrac{\dfrac{\dfrac{[u : y = \mathsf{Fact}(n)]}{y * (n+1) = \mathsf{Fact}(n+1)}}{\exists y(y = \mathsf{Fact}(n+1))}\;\exists^{+}}{(y = \mathsf{Fact}(n)) \rightarrow}\;\rightarrow^{+}_{u}}{\forall y(y = \mathsf{Fact}(n)) \rightarrow \exists y(y = \mathsf{Fact}(n+1))}\;\forall^{+}}{\exists y(y = \mathsf{Fact}(n+1))}\;\exists^{-}}{\exists y(y = \mathsf{Fact}(n)) \rightarrow \exists y(y = \mathsf{Fact}(n+1))}\;\rightarrow^{+}_{v}}{\forall n(\exists y(y = \mathsf{Fact}(n)) \rightarrow \exists y(y = \mathsf{Fact}(n+1)))}\;\forall^{+}$$

$$\dfrac{\exists y(y = \mathsf{Fact}(0)) \qquad \forall n(\exists y(y = \mathsf{Fact}(n)) \rightarrow \exists y(y = \mathsf{Fact}(n+1)))}{\forall n \exists y(y = \mathsf{Fact}(n))}\;\text{up-prim-rec}$$

We name this proof as Proof_fact1. The program extracted from Proof_fact1 is the following:

```
let fun fact n =
  if (n=0) then 1
  else (fact (n-1))*n
```

Let assume to name the base's and step's proofs of Proof_fact1 respectively as $B$ and the step as $S$. We have already seen in section 8.2.2 how to express at programming level, via the Kleene trick, up primitive recursion in terms of up primitive iteration. In the same section we have seen how to do it also at proof level. So replacing $M$ with $B$, $N$ with $S$ and $P(n)$ with $\mathsf{Fact}(n)$ in Figure 8.3, we obtain a new proof, that we name Proof_fact2, with the following computational content:

```
fun fact' n =
 #2(let fun visit m =
      if (m=0) then (0,1)
      else
        let val (j,c)=visit (m-1)
        in (j+1,j*c) end
    in visit n end
```

Proof_fact2 will be a proof with the following shape:

$$\dfrac{\begin{array}{ccc} & & |S \\ & |B & \forall n(\exists y(y = \mathsf{Fact}(n)) \to \exists y(y = \mathsf{Fact}(n+1))) \\ \exists y(y = \mathsf{Fact}(0)) & & |J \\ & |K & \forall^{\mathsf{nc}} n((\exists y(y = n) \ \wedge \ \exists y(y = \mathsf{Fact}(n))) \to \\ \exists y(y = 0) \wedge \exists y(y = \mathsf{Fact}(0)) & & (\exists y(y = n+1) \ \wedge \ \exists y(y = \mathsf{Fact}(n+1)))) \end{array}}{\dfrac{\dfrac{\dfrac{\forall n(\exists y(y = n) \wedge \exists y(y = \mathsf{Fact}(n))) \qquad n}{\exists y(y = n) \wedge \exists y(y = \mathsf{Fact}(n))} \forall^-}{\mathsf{Fact}(n)} \wedge_1^-}{\forall n \mathsf{Fact}(n)} \forall^+} \text{(up-prim-iter)}$$

Where $|K$ and $|J$ can be deduced from Figure 8.3. Now, in section 8.3.2 we have seen how to transform an up primitive iterative proof of the form:

$$\dfrac{\begin{array}{cc} |M & |N \\ P(0) & \forall^{\mathsf{nc}} n(P(n) \to P(n+1)) \end{array}}{\forall n P(n)} \text{(up-prim-iter)}$$

into another proof with an accumulator based extracted program. Now replacing $M$ with $K[B]$, $N$ with $J[S]$ and $P(n)$ with $\exists y(y = n) \wedge \mathsf{Fact}(n)$ in the above schema and then appling the proof transformation described in section 8.3.2 to the proof so instantiated, we obtain a new proof of the formula $\forall n(\exists y(y = n) \wedge \mathsf{Fact}(n))$, that we name Proof_fact3. Thus, from the derivation:

$$\dfrac{\dfrac{\dfrac{\forall n(\exists y(y = n) \wedge \mathsf{Fact}(n)) \qquad n}{\exists y(y = n) \wedge \mathsf{Fact}(n)} \forall^-}{\mathsf{Fact}(n)} \wedge_1^-}{\forall n \mathsf{Fact}(n)} \forall^+$$

we extract the following iterative with accumulator version of the factorial function:

```
fun fact'' n =
 #2(let fun visit m a=
      if (m=0) then a
      else visit (m-1) ((#1a)+1, #1a*#2a)
    in visit n (0,1) end)
```

We would like to point out once more that, even if the program obtained after the application of the above transformation is not particularly complicated, our transformation is completely *automatic* and acts at proof level, that is, the proof itself will constitute a certificate of the correctness of our transformation.

# 9 Conclusions and Future Works

In this thesis we developed a set of proof-transformations in order to extract efficient program from proofs. In the following we will briefly introduce each proof-transformation technique presented and we will discuss possible extensions of it.

**Pruning**
One of the main result in this thesis regarded pruning: we showed on two big examples, the bin packing problem and the perfect matching one, that pruning can be an essential tool to improve the efficiency of the programs extracted from proofs. The aspect that make pruning a proof/program transformation not comparable with other proof/program transformations rely on the fact that pruning *modify* the computational behavior of the extracted programs. This can looks (in a first moment) a property not desideable, but in the truth is the secret of the power of this method: given a proof of a problem with many solutions pruning transform the proof (and so the solution codified in the proof) into another proof, simplifying all the redundant case distinctions.

In chapter 5 then we extended pruning with a more general rule. We proved formally that each simplification that can be done by pruning then is performable by the new pruning rule, and we showed on a case study that the opposite is not true: that is there are simplifications performed by the new rule that is not possible to mimic with pruning.

Further works could regards an extension of the new pruning rule in order to overcame the problem (that we did not treat in our formulation) of pruning as source of inefficiency. In order to make clear this point consider the following example. If we apply pruning on the proof in Figure 9.1 we obtain the proof term:

$$\mathsf{IF}\ \ t_2\ \ (\exists^+\, r_2\, (\mathsf{AX}_1\, u^{t_2}))\ \ (\exists^+\, r_3\, (\mathsf{AX}_2\, u^{\neg t_2}))$$

Now assume in this case that $t_1$ is a fast algorithm, that is that $t_1[x/r]$ can be normalized in just few steps for each input $r$. Suppose further that $t_2$ is very slow. Then we have the following situation: whenever $t_1$ holds, $r_1$ may be immediately returned as the output, but when $\neg t_1$ holds a long computation must be undertaken to determine which of $t_2$ or $\neg t_2$ holds. However, the correctness of the "long computation" does not depend on whether $\neg t_1$ holds. Thus we have a fast way ($t_1$) of discriminating between two ways of computing

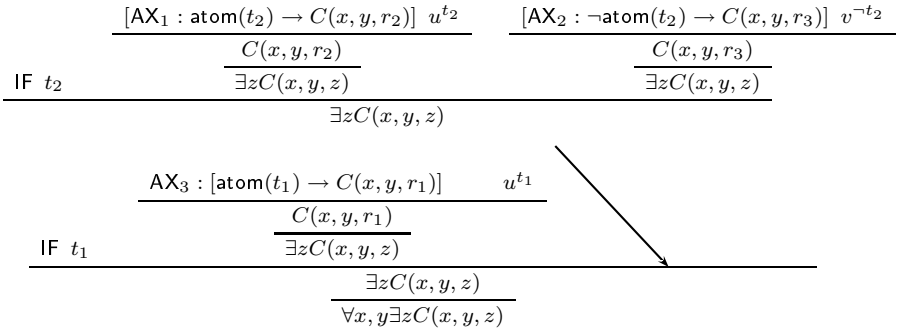



Figure 9.1:

a satisfactory output, one of which is very fast (the simple return of $r_1$) and the other of which is very slow. Further the slow way always works. Pruning in this case has the effect of throwing away the discrimination ($t_1$) and choosing the slow way every time.

**Dynamic Programming**
In chapter 6 we presented an *a doc* proof-transformation in order to synthesize a dynamic program from a constructive proof. The proposed method's name was *list as memory*. The idea consist in evaluating a sufficient amount of data in advance so that the extracted algorithm gets to reuse it instead of recomputing it each time it is needed. This is done introducing in the proof a list of *ad-hoc* axioms. The method we proposed in this thesis can not be applied automatically to an arbitrary proof but it can be seen more as a general schema (that has to be instantiated case by case) to follow in order extract dynamic programs from proofs. Future works in this direction will regards the automation of this process.

**Tail Recursion**
In chapter 7 we have seen how to transform a proof with recursive content into another proof with tail recursive content. We presented two proof transformations: an "accumulator" based one, from which it is possible synthesize the $\Pi$ tail recursive schema and a "continuation" based, from which it is possible to extract the $\Lambda$ schema.

We note that $\Lambda$ is in some way more *general* than $\Pi$. The modification of $\Lambda$ in order to make it working on lists (let us name it $\Lambda_{\mathbf{L}(\rho)}$) instead of naturals is easy; more importantly, the proof from which $\Lambda_{\mathbf{L}(\rho)}$ can be extracted is obtained by a slightly modification of the proof from which $\Lambda$ is extracted. In

the case of lists the end formula to prove should be: $\forall l^{L(\rho)}.(P(l) \to \bot) \to \bot$. Unfortunately we can not extend in the same way $\Pi$ and its proof: $\Pi$ looks intrinsically dependent from the algebra of natural numbers.

Possible applications of $\Lambda$ and $\Pi$ go beyond the *tail recursion*. We noted that there exists proofs from which are extracted programs that run in exponential time that can be turned (by the proofs transformations proposed here) in new proofs from which it is possible to extract polynomial time algorithms. This can appear pretty amazing and we are currently working in order to state such result more precisely.

Another application of the proofs transformations proposed here is an extension of the CPS-transformation over formal proofs (Schwichtenberg [34] and Griffin [19]) but this time concerning the induction axiom. The proposal is to perform CPS over proofs in two stages: a pre-processing step where all the proofs by induction are transformed according to our method, and a second stage where CPS is applied skipping all the proofs by inductions. Currently we are studying also this aspect but it need a deeper investigation.

A final remarks on the formal transformation of Ind_CONT into Ind_ACC presented in section 7.3. It could be interesting to study if, and how, to perform the inverse operation, that is to go from Ind_ACC to Ind_CONT. We argue that it could be done by the *Refunctionalization* technique [14], but also this aspect needs a deeper investigation.

# Bibliography


[1] Penny Anderson. *Program Derivation by Proof Transformation.* PhD thesis, Carnegie Mellon University, 1993.

[2] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

[3] Stefano Berardi. Pruning simply typed lambda terms. *Journal of Logic and Computation*, 6(5), 1996.

[4] Ulrich Berger. Uniform Heyting Arithmetic. *Annals Pure Applied Logic*, 133:125–148, 2005.

[5] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 91–106. Springer Verlag, 1993.

[6] Ulrich Berger, Wilfried Bucholz, and Helmut Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114:3–25, 2002.

[7] Luca Boerio. *Optimazing Programs Extracted from Proofs.* PhD thesis, Computer Science Department of Turin, 1997.

[8] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1), August, 1977.

[9] Luca Chiarabini. Extraction of Efficient Programs from Proofs: The case of Structural Induction over Natural Numbers. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, 2008.

[10] Luca Chiarabini. A new adaptation of the pruning technique for the extraction of efficient program from proofs, 2008. `http://www.mathematik.uni-muenchen.de/~chiarabi/publ.html/PrunInMinlog.pdf`.

[11] K.-M. Chung and H.-I. Lu. An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, 34:373–387, 2004.

[12] Ferruccio Damiani and Paola Giannini. Automatic useless code detection and elimination for hot functional programs. *Journal of Functional Programming*, 10(6), 2000.

[13] Olivier Danvy. Three steps for the cps transformation. *Technical report CIS-92-02*, 1991. DAIMI, Department of Computer Science, University of Århus, Danimark.

[14] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 2008.

[15] Olivier Danvy and Lasse R.Nielsen. Defunctionalization at work. In editor Harald Søndergaard, editor, *Proceedings of the Third International Conference of Principles and Practice of Declarative Programming*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[16] P. Fariselli, M. Finelli, D. Marchignoli, P.L. Martelli, I. Rossi, and R. Casadio. Maxsubseq: An algorithm for segment-length optimization. the case study of the transmembrane spanning segments. *Bioinformatics*, 19:500–505, 2003.

[17] Christopher Goad. Computational uses of the manupulation of formal proofs. Technical report, Stanford Departmet of Computer Science, August 1980. Report No. STAN-CS-80-819.

[18] M.H. Goldwasser, M.-Y. Kao, and H.-I. Lu. Linear-time algorithms for computing maximum-density sequence. *Journal of Computer and System Sciences*, 70(2):128–144, 2005.

[19] Timothy G. Griffin. A formulae-as-types notion of control. In *Proceedings of the 17th Annual ACM Symp. on Principles of Programming Languages, POPL'90, San Francisco, CA, USA*, 1990.

[20] Dan Gusfield. *Algorithms on Strings, Tree and Sequences.* Cambridge University Press, 1997.

[21] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and automatic Program Generation.* Prentice Hall, 1993.

[22] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised$^5$ report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August, 1998.

[23] K. Kent Dybvig. *The Scheme Programming Language.* Mit Press, 1996.

[24] G. Kreisel. Interpretation of Analysis by means of Functionals of Finite Type. In A. Heyting, editor, *Constructivity in Mathematics*, 1959.

[25] Y.-L. Lin, T. Jiang, and K.-M. Chao. Efficient algorithms for locating the length-constrained heaviest segments with applications to biomolecular sequence analysis. *Journal of Computer and System Sciences*, 65:570–586, 2002.

[26] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. Wiley/Teubner Computing Series, 1997.

[27] Zohar Manna and Richard J. Waldinger. Towards automatic program synthesis. *Communications of the ACM*, 14(3), 1971.

[28] Kobayashi Naoki. Type-based useless variable elimination. Technical report, Department of Information Science, University of Tokyo, July 1999. Technical Report 99-02.

[29] Aleksey Nogin. Writing Constructive Proofs Yielding Efficient Extracted Programs. In Didier Galmiche, editor, *Proceedings of the Workshop on Type-Theoretic Languages: Proof Search and Semantics*, volume 37 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000.

[30] Frank Pfenning. Program development through proof transformation. In *Contemporary Mathematics*, volume 106, pages 251–262, 1990.

[31] Dag Prawitz. Ideas and results in proof theory. *Proceedings of the 2. Scandinavian Logic Symposium*, pages 237 – 309, 1971.

[32] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[33] Amr Sabry. Continuations in programming practice: Introduction and survey, 1999. Unpublished manuscript.

[34] Helmut Schwichtenberg. Proofs, lambda terms and control operators. In *Logic of computation. Proceedings of the NATO ASI.Marktoberdorf, Germany*, 1995.

[35] Helmut Schwichtenberg. Programmentwicklung durch beweistransformation: Das Maximalsegmentproblem. In *Bayer. Akad.*, 1996.

[36] Helmut Schwichtenberg. Minimal Logic for Computable Functionals. December 2008.

[37] Helmut Schwichtenberg. Minlog referece manual. `http://www.minlog-system.de/`, December 2006.

[38] Thomas S.Ferguson. Linear programming, a concise introduction. Lecture Notes, `www.math.ucla.edu/~tom/LP.pdf`, 2009.

[39] M.H. Sørensen and P.Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

[40] Volker Sperschneider. *Bioinformatics*. Springer, 2008.

[41] Anne S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2002.

[42] M. Tompa W.L. Ruzzo. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology, ISMB'99*, pages 234–241, 1999.