DERIVING A DYNAMIC PROGRAMMING ALGORITHM FOR BATCH SCHEDULING IN THE REFINEMENT CALCULUS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

İREM AKTUĞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF COMPUTER ENGINEERING

JULY 2003

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Halit Oğuztüzün
Supervisor

Examining Committee Members

Prof. Dr. Faruk Polat

Assoc. Prof. Dr. Ali H. Doğru

Assoc. Prof. Dr. İ. Hakkı Toroslu

Assist. Prof. Dr. Halit Oğuztüzün

Assist. Prof. Dr. Andreas Tiefenbach

ABSTRACT

DERIVING A DYNAMIC PROGRAMMING ALGORITHM
FOR BATCH SCHEDULING
IN THE REFINEMENT CALCULUS

Aktuğ, İrem
MS, Department of Computer Engineering
Supervisor: Assist. Prof. Dr. Halit Oğuztüzün

July 2003, 57 pages

Refinement Calculus is a formalization of stepwise program construction. In this approach a program is derived from its specification by applying refinement rules. The Refinement Calculator, developed at TUCS, Finland, provides tool support for the Refinement Calculus. This thesis presents a case study aiming to evaluate the applicability of the theory and the performance of the tool. The Refinement Calculator is used for deriving a dynamic programming algorithm for a single-machine batch scheduling problem. A quadratic algorithm is derived by refining a formal specification of this problem into executable code. The need for stronger support for relevant domain theories and abstraction mechanisms in the target language have been noted.

Keywords: Formal Methods, Refinement, Program Synthesis

ÖZ

TOPLU İŞ ÇİZELGELEME İÇİN BİR DİNAMİK PROGRAMLAMA ALGORİTMASININ İNCELTME KALKÜLÜSÜNDE TÜRETİMİ

Aktuğ, İrem
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Assist. Prof. Dr. Halit Oğuztüzün

Temmuz 2003, 57 sayfa

İnceltme Kalkülüsü adım adım program oluşturan bir formalizasyondur. Bu yaklaşımda, bir program belirtiminden inceltme kuralları uygulanarak türetilir. İnceltme Kalkülatörü TUCS, Finlanda'da geliştirilmiş olup, İnceltme Kalkülüsü'ne araç desteği sağlamaktadır. Bu tez teorinin uygulanabilirliğini ve aracın performansını ölçmeyi amaçlayan bir örnek olay incelemesi sunmaktadir. İnceltme Kalkülatörü tekmakinada toplu iş çizelgeleme için dinamik programlama algoritması türetiminde kullanılmıştır. Kuadratik bir algoritma bu problemin formal belirtiminin çalıştırılabilir koda inceltilmesiyle türetilmiştir. İlgili tanım bölgesi teorilerine ve hedef dilde soyutlama mekanizmalarına daha fazla desteğin gereği farkedilmiştir.

Anahtar Kelimeler: Formal Metodlar, İnceltme, Program Sentezi

To my grandmothers

ACKNOWLEDGMENTS

I owe my deepest gratitude to my supervisor - Halit Oğuztüzün. Despite his busy schedule, he has always managed to find time for discussions on various aspects of my research. Without his continuous encouragement and friendly support combined with invaluable expert advice, this thesis would have never been finished.

I would like to thank Linas Laibinis from TUCS, Finland, for providing me the software, and giving continuous support throughout this study.

I would like to thank our system administrators Ersan and Barış for helping me install the software and always being there when I need help.

I thank my parents for their ongoing support despite my ongoing mistakes.

I thank all my friends who have cheered me up through phone calls, chats and e-mail whether they be in the room, or in Canada.

Finally, my admiration goes to my love, Barış, for the happiness that he has given me. I will remember a lifetime and more.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.1 Refinement Calculus

Formal methods emerged as a result of the failure of traditional testing to prevent software errors. They aim to guarantee error-free software by proving mathematically that the software being developed will function as expected. Furthermore, these expectations are expressed as mathematical expressions. This adds to the predictability of the system, since mathematical formulation requires more precision as to requirements expressed in natural language.

Program verification and program refinement are two different methods to develop correct software. In verification, a complete program is proven to accomplish what was expected by the specification. Whereas, in the refinement approach, the specification is transformed into a correct implementation. The result of each refinement step is proved to be consistent with the previous program and this can be traced back to the initial specification.

Stepwise program refinement was formalized into the Refinement Calculus by Back [4, 3]. He introduced the central notions of the calculus like the refinement relation between programs and using specifications as program statements. Morgan extended the calculus by miraculous statements and published an influential book that established the applicability of the calculus to practical programming problems by presenting a number of case studies. Back and Wright published the complete higher-order logic formalization of the calculus in [5].

I.2 Refinement Calculator

Even for small programs, the proofs involved in derivations are often long and involved. Refinement Calculator was developed at TUCS, Finland [6, 7] to aid the user with program construction proofs. It records proofs and ensures the soundness of refinement steps. The refinement calculus theory was written in the proof assistant HOL [2] as a theory. The calculator provides a user-friendly interface which is comprised of options linked to functions implemented as a part of this theory.

I.3 Thesis Overview

In this thesis, our aim is to derive a dynamic programming algorithm for a one machine batching problem using the Refinement Calculator. Our motivation is to evaluate performance of the tool and the approach in this derivation effort.

We have used the the Refinement Calculator for deriving a dynamic programming algorithm for a single-machine batch scheduling problem. First, a formal specification of this problem is formulated. Then, the specification is refined by applying to it the rules available in the Calculator. As a result, executable code version of a quadratic algorithm has been derived. The need for stronger support for relevant domain theories and abstraction mechanisms in the target language have been noted.

The rest of the thesis is organized as follows. Chapter II gives some background on refinement calculus theory, especially on the refinement relation. Then the Refinement calculator is introduced. Its interface, the syntax of the language that it provides and the rules that have been used in the scope of the study are briefly explained. Some details of the array theory, which has been used in the study, are also presented in this chapter.

Chapter III begins with the statement of the batch scheduling problem and the algorithm. It goes on to explain some general points about the derivation. Next, the derivation phase is explained in two stages. Each stage result in a the part of the aimed algorithm.

A mini dictionary of user-defined predicates and functions including the HOL definitions are given in Appendix B. A list of theorems that have been used is given in Appendix C.1.

Chapter IV concludes with discussion on implementational aspects of the tool. Pos-

sible future work on deriving algorithms for the particular problem is suggested in the Future Work section.

CHAPTER II

BACKGROUND

II.1 Refinement Calculus

The stepwise refinement method for construction achieves the derivation of an executable program from its specification by a series of transformations (refinements). Refinement calculus is a formalization of the method of stepwise refinement to construct programs. Refinement is a preorder relation between programs:

$$S_0 \sqsubseteq S_1 \sqsubseteq ... \sqsubseteq S_n$$

Here S_0 is being refined by S_1 and so on. This sequence establishes $S_0 \sqsubseteq S_n$ because of transitivity.

Refinement can be used to achieve different purposes. S_0 may represent a specification and S_n an executable code, or S_0 may be a program which is in NP while S_n is one in P. Thus program construction and optimization can both be considered as refinements.

The calculus is based on Dijkstra's weakest precondition semantics for programs [8]. The meanings of program statements are defined using predicates over program states (given by the value assignments to program variables). The weakest precondition semantics defines the weakest initial predicate (a precondition) from which it is guaranteed that the intended result (a postcondition) is reached by the given statement. Therefore, program statements are modeled as functions that map postconditions to preconditions.

The refinement relation is defined in terms of weakest preconditions of the related programs. The weakest precondition of program S with respect to postcondition P (wp(S, P)) is a predicate that selects all the initial states from which it is possible to reach, running S, the states selected by P, provided S terminates.

Program S_0 is refined by S_1 iff S_1 establishes every postcondition P that is established by S_0:

$$\forall \mathrm{P}.\ \mathrm{wp}(S_0,\ \mathrm{P}) \rightarrow \mathrm{wp}(S_1,\mathrm{P})$$

The above formula, states that S_1 takes all the states that S_0 takes to P, and possibly more. The rules of refinement are extracted using this underlying definition.

A program is almost never refined as a whole, which would be very complex. Instead, each refinement can be applied on some component of the program and replaces this component with a refined one. The correctness of the whole program is guaranteed in such a case, since all statement constructors of the refinement calculus language are monotonic with respect to refinement. For example, we can show the following holds:

$$\frac{S_1 \sqsubseteq S_1' \wedge S_2 \sqsubseteq S_2'}{S_1; S_2 \sqsubseteq S_1'; S_2'}$$

i.e. sequential composition is monotonic in both arguments. Using this property, one can advance by focusing on small components and refining them in isolation from the context.

II.2 Refinement Calculator

The Refinement Calculator has been developed at TUCS, Finland [6, 7] to aid users with program construction proofs in Refinement Calculus. Since the logical basis of the refinement calculus is a classical higher-order logic, a tool could be built on any such environment. HOL [2] was chosen as the base because it supplies the means for a transformational reasoning style and subderivations involved in refinement proofs. In order to provide a more user-friendly environment, the calculator has been built upon a general graphical user interface that communicates with HOL itself, TkWinHOL [10].

To make refinement proofs, the refinement calculator theory extension is loaded onto TkWinHOL. Other extensions make it possible to carry on correctness and data refinement proofs, work with procedures, with general logic and lattice theories, etc.

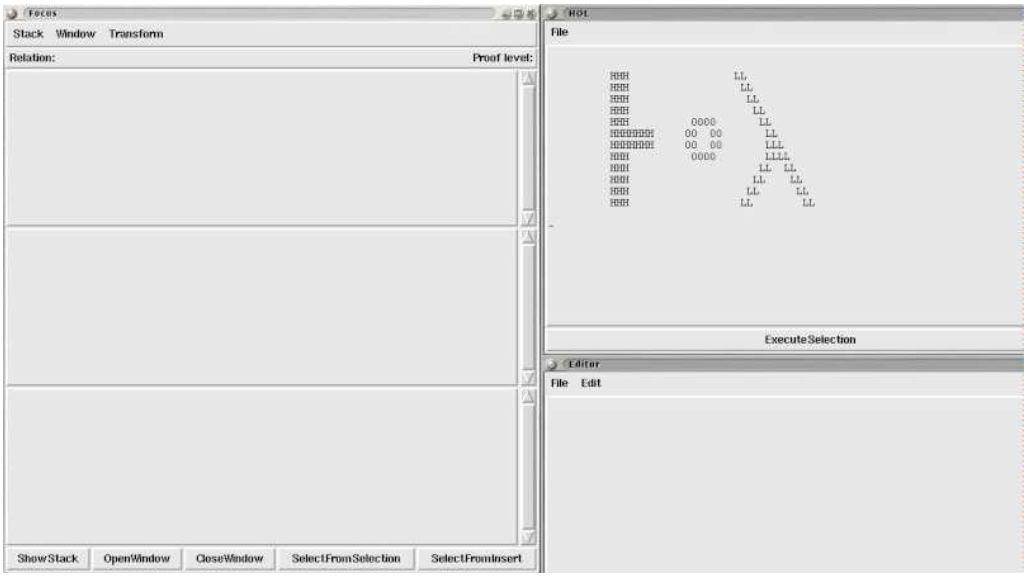

Figure II.1: The Refinement Calculator Interface

The interface (II.1) has three windows. The Focus window is the one where the derivation occurs, and is composed of 3 subwindows. The top subwindow contains the term that is the current focus. This can be the whole proof statement or any subterm of it. The middle subwindow lists context information, conjectures (proof obligations), and lemmas (statements of conjectures that have been proved). The bottom subwindow shows the current window theorem. The interface also contains the HOL window for executing HOL commands, loading extensions or previous work in the form of sml files, and the Editor window consisting of a simple text editor.

The TkWinHOL interface specifically works with window interface. The work involved in proofs usually consists of selecting some information on the Focus window by clicking, choosing a transformation rule from the pull-down menus to apply to the selection, and possibly entering various arguments needed by the rule using the keyboard. For example, for the Cond Introduction rule (see below) the guard condition G is entered from the keyboard. The application of a rule may give rise to proof obligations that must be proved for the derivation to be complete.

For making subderivations, it is possible to focus on subterms before applying rules on them. This means that only this specific subterm is seen in the top subwindow. Surrounding context is hidden, except the listing of the assumptions that it allows, in the middle subwindow. These assumptions can also be highlighted in order to be used

in transforming the focus. After application of desired rules to the term in focus have been completed, the previous focus can be reached by closing the selection.

When the derivation is finished, the result is a conditional theorem of the form : $\Phi \vdash S_0 \sqsubseteq S_n$ in the HOL logic. In this theorem, S_0 is the original specification, S_n a refinement of it and Φ is a sequent with undischarged proof obligations as assumptions. If all of the obligations that has arised in the course of the derivation have been proved, Φ is empty and the result is a theorem. Each refinement done on a subterm is reflected on the whole program.

$$\frac{T \sqsubseteq T'}{S_i[T] \sqsubseteq S_i[T']}$$

II.3 The Syntax of the Language

The theory focuses on imperative state-based programs. So the basic programming constructs of this paradigm are included in the language of the tool. The syntax of the language of the tool appears below.

$Prog$ ::= `program` $Name$ `var` v: $Type$ `.` Com

Com ::= Com;Com (sequential composition)

| $\{BooleanTerm\}$ (assertion)

| v `:=` $Term$ (assignment)

| v `:=` v'`.` $BooleanTerm$ (nondeterministic assignment)

| `do` $BooleanTerm \rightarrow Com$ `od` (while-do statement)

| |[`var` v: $Type$ `.` Com]| (block with local variable)

Here variable name v may represent a list of variable names. $Type$ denotes any HOL type, $Term$ represents any HOL term, and $BooleanTerm$ represents any boolean-valued HOL term.

The interesting part of the syntax is the abstract statements it allows for specification purposes. In this language, assertions and the nondeterministic assignments comprise the precondition and postcondition respectively.

The assertion is a boolean term that is expected to hold in the program execution at that point. It usually contains conjuncts making some statement about the current value of a program variable. The nondeterministic assignment is the expression of a parallel assignment to the list of variables on the left hand side, the corresponding values on the right hand side respectively. These values are represented by the primed version

of the corresponding variable and they satisfy the postcondition that appears after the dot. The postcondition may also contain unprimed versions of the variables on the left hand side, which signify the values of those variables before the assignment. The nondeterministic assignment lets us set some condition on the final values of variables without determining the exact value.

II.4 Extensions

The extensions of the Calculator that have been used in the scope of this study are Refinement, Correctness, and General Logic. Each extension is loaded to produce a pull down menu of choices in the Focus window. In addition to these commands, the options that the interface offers in the Transform menu are also used.

Of these extensions, only Refinement and Correctness are based on Refinement Calculus and are explained in detail below. The Transform menu offers general rules like Beta Conversion, and rules to transform the focus using some theorem (e.g. Rewrite). The General Logic menu adds to this list logical options like universal elimination, existential introduction, distribution and transitivity.

II.5 Refinement Extension

For a comprehensive list of refinement rules offered in The Refinement Calculator, please see the appendix of [11]. The most important rules as used in the scope of this study are presented below. The first four rules are for introducing program constructs. The next two serve to propagate context information in assertions. The last one is for adding specification constants.

Add Assignment

$$\frac{\mathtt{x} \text{ not free in } post}{\vdash \mathtt{v} := \mathtt{v}' \,.\, post \sqsubseteq \mathtt{x} := \mathtt{E}; \mathtt{v} := \mathtt{v}' \,.\, post}$$

A leading assignment is added before an assignment statement.

Example:

```
fsum,c,k:= fsum',c',k'.(fsum'= (sigma 1 n)) /\
                       (contains_batch_cost c' n s)
```

$\sqsubseteq$

```
k,fsum:=n, 1;
fsum,c,k:= fsum',c',k'.(fsum'= (sigma 1 n)) /\
                       (contains_batch_cost c' n s)
```

Cond Introduction

$$\vdash S \sqsubseteq \texttt{if}\ G\ \texttt{then}\ S\ \texttt{else}\ S$$

The conditional construct is introduced.

Example:

```
costn:= update costn k (minvalue2 k m)
```

$\sqsubseteq$

```
if ((lookup costn m) < (lookup costn k))
then
    costn:= update costn k (minvalue2 k m)
else
    costn:= update costn k (minvalue2 k m)
```

Block Introduction

$$\frac{\texttt{x}\text{ not free in }post}{\vdash \{pre\}; \texttt{v} := \texttt{v}' \,.\, post \sqsubseteq |[\texttt{varx} : T \,.\, pre; \texttt{v}, \texttt{x} := \texttt{v}', \texttt{x}' \,.\, post]|}$$

A new variable x of type T is introduced to the environment.

Example:

```
{ ((asize c) = (n + 1)) };
fsum,c:= fsum',c'.(fsum'= (sigma 1 n)) /\
                  (contains_batch_cost c' n s)
```

$\sqsubseteq$

```
|[var k:num.
   { ((asize c) = (n + 1)) };
   fsum,c:= fsum',c'.(fsum'= (sigma 1 n)) /\
                     (contains_batch_cost c' n s)
]|
```

Loop Introduction

$$\frac{\begin{array}{c}\vdash pre \Rightarrow inv \\ \vdash (inv/\backslash G/\backslash t = e) << Body >> (inv/\backslash 0 \leq t/\backslash t < e) \\ \vdash (\neg G/\backslash inv) \Rightarrow post[\texttt{v}' := \texttt{v}] \\ \texttt{v}\text{ not free in }post\end{array}}{\{pre\}; \texttt{v} := \texttt{v}'.post \sqsubseteq \texttt{do}\ G\ \rightarrow\ Body\ \texttt{od}}$$

Here t is the termination function or the variant. While the invariant inv must be preserved, the variant t is required to decrease. The variant is essential so as to guarantee the termination of the loop in cases where the guard G never becomes false.

In the Calculator, after the execution of the Loop Introduction command, the three proof obligations for the refinement appear in the middle window, under the title "Conjectures". They may be discharged or left as is, in which case they will be added as assumptions to the resulting refinement theorem. In order to discharge these assumptions, subderivations may be started by selecting a proof obligation from the middle window, and choosing Establish from the Windows menu. This will bring the proof obligation into Focus and hide the refinement proof until the subderivation is completed.

In figures (II.2) and (II.3), the top portion of the Focus window is shown before and after loop introduction.

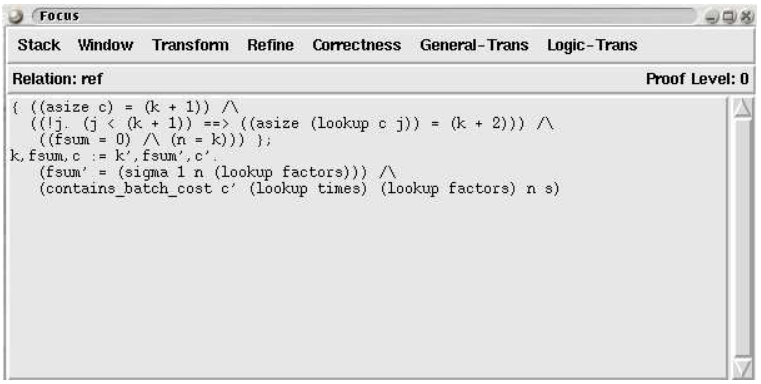

Figure II.2: Top subwindow before loop introduction

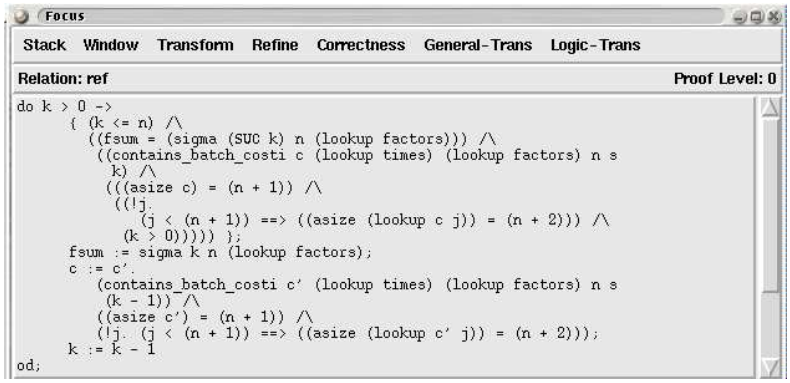

Figure II.3: Top subwindow after loop introduction

Example:

```
guard: k > 0
body: fsum := (sigma k n);
      c := c'.(contains_batch_costi c' n (k-1));
      k := k-1
invariant:
      (k <= n) /\
      (fsum= (sigma (SUC k) n)) /\
      (contains_batch_costi c n k)
variant: k
```

Conjecture 1:

```
var k:num,fsum:num,n:num,c:num.
(!s0.  (!n0.  (!x.
(n = n0) /\
(k <= n) /\
(fsum = (sigma (SUC k) n)) /\
(contains_batch_costi c n k) /\
((k > 0) /\ (k = x))
<< fsum := sigma k n;
   c := c'.(contains_batch_costi c' n (k - 1));
   k := k - 1 >>
(n = n0) /\
(k <= n) /\
(fsum = (sigma (SUC k) n)) /\
(contains_batch_costi c n k) /\
(k < x))))
```

Conjecture 2:

```
(!n0.  (!c0.  T ==>
(n0 <= n0) /\
(0 = (sigma (SUC n0) n0))) /\
(contains_batch_costi c0 n0 n0))
```

Conjecture 3:

```
(!n0.  (!k0.  (!c0.
(~ (k0 > 0)) /\ (k0 <= n0) /\
(contains_batch_costi c0 n0 k0) /\
 ==>
((sigma (SUC k0) n0) = (sigma 1 n0)) /\
(contains_batch_cost c0 n0 s0))))
```

Refinement:

```
{ ((fsum = 0) /\ (n = k)) };
k,fsum,c := k',fsum',c'.
             (fsum' = (sigma 1 n)) /\
             (contains_batch_cost c' n)
```

$\sqsubseteq$

```
do k > 0 ->
   { (k <= n) /\
   (fsum = (sigma (SUC k) n)) /\
   (contains_batch_costi c n k) /\
   (k > 0) };
   fsum := sigma k n;
   c := c'.(contains_batch_costi c' n (k - 1));
   k := k - 1
od;
   { (k <= n) /\
   (fsum = (sigma (SUC k) n)) /\
   (contains_batch_costi c n k) /\
   (~ (k > 0)) }
```

Push Assert

$$\{pre\} \ ; \ \texttt{v:=}\ E \sqsubseteq \texttt{v:=}E; \ \{pre \wedge \texttt{v=E}\}$$

or

$$\{pre\} \ ; \ \texttt{if}\ G\ \texttt{then}\ S\ \texttt{else}\ T \sqsubseteq \texttt{if}\ G\ \texttt{then}\ \{pre \wedge G\};\ S\ \texttt{else}\ \{pre \wedge \neg\ G\};T$$

This rule propagates an assertion forward. The target is a sequential composition of an assert and assignment/conditional.

Example to the first rule:

```
{ ((asize c) = (n + 1)) };
k,fsum := n, 0
```

$\sqsubseteq$

```
k,fsum := n, 0;
{ ((asize c) = (n + 1)) /\ (fsum = 0) /\ (k = n) }
```

Drop Assert

$$\{\text{pre}\} \ ; \ \text{S} \sqsubseteq \text{S}$$

or

$$\text{S}; \ \{\text{pre}\} \sqsubseteq \text{S}$$

This rule drops an assertion, provided that the target is a sequential composition, with an assertion as the first or last component.

Example to the first rule:

```
{ (k <= n) /\ (k > 0)
  (fsum = (sigma (SUC k) n)) /\
  (contains_batch_costi c n s k) };
  fsum := sigma k n
```

$\sqsubseteq$

```
fsum := sigma k n
```

Use Specification Constant

The argument of this rule t is of the form of $(x = x_0) \wedge (y = y_0) \wedge \ldots$ where $x, y, \ldots$ are program variables and $x_0, y_0, \ldots$ are fresh variables (specification constants). The new focus has the form $\{(x = x_0) \wedge (y = y_0) \wedge \ldots\}; S$ where S is the old focus. The aim is to transform this new focus into a focus S' which does not mention the specification constants. At this point Close Window closes the subderivation and transforms the original focus S into S'. The inference rule justifying this operation is

$$\frac{\Phi \vdash \{x = x_0\}; S \sqsubseteq S'}{\Phi \vdash S \sqsubseteq S'}$$

provided x_0 does not occur free in Φ, S or S' (the rule of specification constant elimination).

Example: The specification constant `NEXT` is introduced.

```
next := next'. (!i. ((k < i) /\ (i < (asize next'))) ==>
                    ((lookup next' i) = (lookup next i)))
```

$\sqsubseteq$

```
{next = NEXT};
next := next'. (!i. ((k < i) /\ (i < (asize next'))) ==>
                    ((lookup next' i) = (lookup next i)))
```

II.6 Correctness Extension

Note that a term of form *pre* << P >> *post*, as used in the Loop introduction rule is used to denote a total-correctness assertion, meaning that the program P is guaranteed to establish the postcondition (*post*), when executed in an initial state satisfying the precondtion (*pre*). These assertions correspond to usual verification of a program segment.

II.7 Array Theory In The Refinement Calculator

The array theory includes the definitions of a new type `array`, and functions that operate on the values of this new type. These functions serve to: query the size of a given array (`asize`), access the value of an element (`lookup`), and change the value of an element (`update`).

Variables of type array contain values of type array. A value of type array is determined by its size and lookup function. Changing the element of an array corresponds to modifying its lookup function. Therefore, the usual notation of selectively updating the elements is not used, instead the updated value of the array variable is assigned to itself. Instead of array[i] := 5, `array := (update array i 5)` is used.

The elements of an array of size n are referred to by using indexes from 0 to n-1. Updates to indexes that are out of bounds return undefined array values, and similarly lookups return arbitrary values.

The array theory is loaded to the Calculator as a separate file.

II.8 Related Work

Since the formalization of the stepwise refinement method into the refinement calculus [3], the theory has been extended on various sides. First, data refinement techniques have been studied [15, 16]. In recent years, the theory has been applied for the stepwise derivation of parallel and reactive programs [12], [13], object-oriented programs [14], and probabilistic programs [17]. The case study undertaken in this thesis, however, does not require the use of such extensions.

As the development of the Refinement Calculator is quite recent, case studies on it are rare. The examples of finding the maximum element of an array in [6], and array sorting in [6] and [9] are the only ones to be found in the literature.

CHAPTER III

DERIVATION

III.1 The Problem

In the batch scheduling problem, there is a queue of N jobs that are to be processed in that same order, on one machine. The queue must be partitioned into batches and a partition that has the minimum overall cost is sought. For each job j, its execution time (t_j) and cost factor (f_j) are given. There is a constant batch startup time S. All the jobs in a batch are output at the same time, after all of them have been executed. To calculate the cost of job j, the time elapsed since the start of the first batch to the completion of the jobs in the batch containing j, is multiplied with the cost factor of job j.

The following example will be used for demonstration throughout this chapter. Assume a queue of 6 jobs, with execution times (4, 2, 2, 1, 1, 3), and cost factors (1, 5, 3, 2, 1, 2), which are to be run on a machine with setup time S=1. If the jobs are partitioned into the three batches {1, 2}, {3}, {4, 5, 6}, then the output times of the jobs will be (7, 7, 10, 16, 16, 16), and the cost of the jobs will be (7, 35, 30, 32, 16, 32), respectively. The overall cost of the partition is the sum of the costs of the jobs, 152.

Supposing it contains the jobs k through m, the processing time of the b^{th} batch equals:

$$time_b = S + \sum_{j=k}^{m} t_j \tag{III.1}$$

And the cost of job j at b^{th} batch equals:

$$cost_j = f_j * \sum_{v=1}^{b} time_v \tag{III.2}$$

Finally, the overall cost of a partition with K batches is calculated using the formula:

$$overallcost = \sum_{b=1}^{K} (\sum_{j=i_b}^{N} f_j)(S + \sum_{j=i_b}^{i_{b+1}-1} t_j) \tag{III.3}$$

where i_b represents the first job of batch b and so the batch b contains the jobs i_b through $i_{b+1} - 1$. (i_{K+1} is set to *N+1* by convention)

Albers and Brucker (1993) reduce this problem to a shortest path problem in directed graphs: The edges represent the batches, and the length of an edge represents the cost of running the corresponding batch. Thus, each partition is represented by a path from the first job to the $(N+1)^{st}$, which is inserted as a placeholder.

Let F_j be the length of a shortest path from j to *N+1*, $F_j(k)$ be the length of a shortest path from j to *N+1* which contains *(j,k)* as the first edge, and c_{ij} be the length of the edge from i to j.

Then they observe:

$$c_{ij} = (\sum_{v=i}^{N} f_v)(S + \sum_{v=i}^{j-1} t_v) \tag{III.4}$$

$$F_j(k) = c_{jk} + F_k \tag{III.5}$$

$$F_j = min\{F_j(k) | j < k \leq N+1\} \tag{III.6}$$

The cost of the path starting from the first node (F_1) is the required result.

It is possible to extract the following dynamic programming algorithm from this formulation as a solution of the shortest path problem:

```
FactorsSum := 0
for i=N downto 1
    FactorsSum := FactorsSum + factors[i];
    TimesSum:=0;
```

```
    for j=i+1 to N+1
        TimesSum := TimesSum + times[j-1];
        cost[i][j] := FactorsSum * (S + TimesSum)
    endfor
endfor

F[N+1] :=0;
for i=N downto 1
    F[i]:= cost[i][i+1] + F[i+1];
    next[i]:=i+1;
    for j=i+2 to N+1
        if (F[i] > cost[i][j]+F[j])
        then F[i]:= cost[i][j]+F[j];
            next[i]:=j;
        else skip;
        endif
    endfor
endfor
```

Arrays *factors* and *times* are of size $(N+1)$ which are the input of the algorithm. Initialization for the arrays cost, F and *next* are unnecessary since any element of these arrays that is accessed is computed by the algorithm beforehand.

The first part of the algorithm computes the cost of a batch containing the jobs i through *j-1* using (III.4). In the second part, the minimum cost of reaching from i to j is computed using the same information for *i+1* to j, *i+2* to j and so on. The first edge in the minimally costed path is stored , in addition to the cost of this path. The node that is visited next in the path is stored in the *next* array, while the corresponding cost is stored in the F array. The running time of this algorithm is $O(N^2)$.

III.2 General Points

Prior to deriving the algorithm, it is necessary to write a formal specification of the corresponding part of the problem as a HOL expression. These specifications usually

consist of a precondition in the form of an assertion and a postcondition in the form of a nondeterministic assignment, and are fed to the Refinement Calculator using the Begin Derivation command.

To keep the specification concise, some auxiliary predicates/functions have been defined. These definitions are recorded in a file that is loaded to the calculator before any derivation begins (See B for HOL definitions). The definitions are expanded at the points of use.

Partitions have been represented as arrays of type `num`. (Although in the problem statement, the term "partition" has been used for a sequence of batches that cover all the jobs in the queue, from now on the term will have a broader meaning that will include partitionings of jobs starting from any given job in the queue) If there are k batches in a partition, the array representing the partition is of size $(k+2)$.

In each index position from 1 to k, the first job of some batch is stored. Notice that the sequence of numbers stored in this range is strictly increasing. For any two jobs i and j, if $i < j$, i has to run before j and so can not be in a batch that runs after the batch containing j. The last element of the array is always $N+1$, which is a dummy job to mark the end of the last batch. The 0^{th} element of the array is never used by convention. The example partition given in the previous section that consists of the batches {1,2},{3},{4,5,6}, would be represented with the following array:

0	1	2	3	4
X	1	3	4	7

Figure III.1: Example partition

The specification of our problem can be formulated as below:

```
var times:(num) array; factors:(num) array; s:num; n:num;
next:(num) array; costn:(num) array.
{ (n >= 1) /\
((asize next) = (n+1)) /\
((asize costn) = (n+2)) };
next,costn := next', costn'.
(?(p:(num) array). (contains_partition next' 1 n p) /\
 (sub_min_cost_p  (lookup times) (lookup factors) 1 n s p) /\
( (lookup costn' 1) =
    (overallcost  (lookup times) (lookup factors) n s p) )
)
```

The arrays `times` and `factors` are the arrays that contain the execution times and cost factors for each of the `n` jobs. The batch startup time is `s`. The arrays `next` and `costn` (which correspond to *next* and F in the forementioned algorithm) will contain the desired partition and its cost after the execution of the program.

The precondition states that there is at least one job in the queue and the sizes of `next` and `costn` are assumed to be `(n+1)` and `(n+2)`, respectively.

The postcondition states that of the declared variables, only the values of the arrays `next` and `costn` will change. A partition `p` will be created as a result. The conjunct containing the predicate `contains_partition` requires that this partition be encoded in the final value of `next`. This partition is to include all the jobs from 1 to `n`, and to be the minimum-costed among all such partitions by the definition of the `sub_min_cost_p` predicate. The function `overallcost` computes the cost of a partition. The last conjuct requires that the cost of partition `p` be assigned to the first position of array `costn`.

The algorithm has been derived in two stages, due to the limitations of the Calculator. For these parts, two different specifications have been formulated, which can be sequentially composed to give the specification above.

In the first stage, the edge lengths (c_{ij}'s) in Albers and Brucker's algorithm are calculated. These correspond to the cost of a single batch specified by its first and last elements. The second stage computes, for every job j, the cost of the minimum costing partition of jobs from j to N, and the first job of the second batch in such a partition. The resulting partition containing all the jobs can be extracted from this computation.

These separate programs are united into a single one simply by sequential composition. The first program outputs an array that contains the individual batch costs, while the second one uses the entries of this array as edge lengths when computing the cost of a path. The second program produces a next batch array which implicitly contains the resulting partition and another array which contains its cost.

III.3 The Cost of Individual Batches

The specification for this part is:

```
    program costeval
    var times:(num) array; factors:(num) array; s:num; n:num;
    c:((num) array) array.
{ ((asize c) = (n+1))  /\
```

```
  (!j. ((j < (n+1)) ==> ((asize (lookup c j)) = (n+2)))) };
c:= c'. (contains_batch_cost c'
                  (lookup times) (lookup factors) n s)
```

The `contains_batch_cost` predicate selects 2D arrays such that when $i < j$, the $array[i][j]$ holds the cost of the batch consisting of the jobs i to $j-1$. Since the jobs are numbered from 1 to `n`, `c` should contain `n+1` rows , while there are `n+2` columns, since the cost of a batch that contains the $\mathtt{n}^{th}$ job would be stored in the $(\mathtt{n}+1)^{th}$ column. The array does not need any initialization, since all the entries that will be used in the second program are updated in this program, one by one.

As per equation (III.4) for every batch cost, two sums are to be computed: one for execution times, the other for cost factors. In order to keep the algorithm $O(N^2)$ these two should be computed incrementally. Therefore, two loops are needed for filling `c`: the outer loop calculates the first sum which depends on i only, while the inner one calculates the second sum which depends on both i and j. These two sums are kept in the variables `fsum`, and `tsum` respectively; the product thereof is assigned to `c[i][j]` (see figure III.2).

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	X	X	X	X	X	X	X	X
1	X	X	70	98	126	140	154	196
2	X	X	X	39	65	78	91	130
3	X	X	X	X	24	32	40	64
4	X	X	X	X	X	10	15	30
5	X	X	X	X	X	X	6	15
6	X	X	X	X	X	X	X	8

Figure III.2: Array `c` after the execution of the first stage of the algorithm

When refining, the outer loop is introduced first. As a preparation, a loop variable `k` and `fsum` are to be introduced. Multiple variable introduction causes problems in the Calculator when using these variables in loops, so the variables have to be introduced by separate blocks. In order to introduce a variable, first the assertion-assignment pair is brought to focus. Since assertions can not be carried into (push through) blocks and

loops by the Calculator, the assertion about the size of c should be stated explicitly at the introduction of the block. Block Introduction from the Refinement menu is chosen, and the below is entered as argument.

```
|[var fsum:num.
{ ((asize c) = (n + 1)) /\
   (!j. (j < (n + 1)) ==> ((asize (lookup c j)) = (n + 2))) };
fsum,c:= fsum',c'.(fsum'= (sigma 1 n (lookup factors))) /\
(contains_batch_cost c' (lookup times) (lookup factors) n s)
]|
```

This refinement preserves the information in the previous focus. Additionally, it states that a new variable (**fsum**) has been added to the environment and its value will be changed to the sum of the elements of array factors as a result of the nondeterministic assignment. The function **sigma** aims to implement the $\sum$ operator. It takes two **num** type arguments for the lower and upper bounds of the sum and one $num \rightarrow num$ type function argument, for which the sum will be calculated. In this case, **fsum** will be equal to $\sum_{v=1}^{N} factors[v]$.

The loop variable **k** is introduced in the same manner, by first focusing on the current assertion-assignment pair and then choosing Block Introduction with the below block as argument.

```
|[var k:num.
{ ((asize c) = (n + 1)) /\
   (!j. (j < (n + 1)) ==> ((asize (lookup c j)) = (n + 2))) };
fsum,c,k:= fsum',c',k'.(fsum'= (sigma 1 n (lookup factors))) /\
           (contains_batch_cost c' (lookup times) (lookup factors) n s)
]|
```

Although no condition on the final value of **k** is included in the nondeterministic assignment, it is necessary to state that **k** will have some (possibly different) value **k'** at the end of the assignment, in order to enable any change to the value of **k** in further refinements of this assignment.

Next **k** and **fsum** need to be initialized to **n** and 0 respectively. For this, we focus on the nondeterministic assignment. Next, Add Assignment from the Refinement menu is chosen and **k,fsum:=n, 1** is entered as argument. As a result, this parallel assignment is added to the program, above the nondeterministic assignment statement. The assignments do not have to be in parallel. This feature merely saves the user from making consecutive Add Assignment refinements. The loop which is introduced after

these initializations has to access the information about the values of these variables after the initialization. The present assertion should be extended to contain this new information. For this purpose, the assertion-initialization pair is brought to focus and Push Assert from the Refinement menu is chosen.

Before the introduction of the loop, the program is as below from the highest proof level:

```
program costeval
var times:(num) array; factors:(num) array; s:num; n:num;
c:((num) array) array.
|[var fsum:num.
  |[var k:num.
    k,fsum := n, 0;
    { ((asize c) = (n + 1)) /\
      (!j. (j < (n + 1)) ==> ((asize (lookup c j)) = (n + 2))) /\
      (fsum = 0) /\ (k = n) };
     k,fsum,c := k',fsum',c'.
     (fsum' = (sigma 1 n (lookup factors))) /\
     (contains_batch_cost c' (lookup times) (lookup factors) n s)
  ]|
]|
```

In order to introduce the loop, first we focus on the assertion and nondeterministic assignment pair. Then, Loop Introduction is chosen from the Refinement menu with the following arguments:

```
guard: k > 0
body: fsum := (sigma k n (lookup factors));
      c := c'. (contains_batch_costi c'
                    (lookup times) (lookup factors) n s (k-1) ) /\
                ((asize c') = (n + 1)) /\
               (!j. (j < (n + 1)) ==>
                        ((asize (lookup c' j)) = (n + 2)));
      k := k-1
invariant: (k <= n) /\
      (fsum= (sigma (SUC k) n (lookup factors))) /\
      (contains_batch_costi c
             (lookup times) (lookup factors) n s k) /\
      ((asize c) = (n + 1)) /\
      (!j. (j < (n + 1)) ==> ((asize (lookup c j)) = (n + 2)))
variant: k
```

The predicate `contains_batch_costi` is has the same definition with `contains_batch_cost` predicate but has a smaller scope, testing if the rows from `k`

to n (where k is the last argument to the predicate) contains the corresponding batch costs.

The nondeterministic assignment to c in the body states that at every execution, the range of the part of c that contains correct values will be extended by one row, and in the meanwhile, the size of c will be preserved. By the guard, the last execution will occur when k equals 1 and so the first row will be updated last.

Three conjectures are produced by the introduction of this loop.

```
((k <= n) /\
(fsum = (sigma (SUC k) n (lookup factors))) /\
(contains_batch_costi c (lookup times)
                    (lookup factors) n s k) /\
((asize c) = (n + 1)) /\
(!j.  (j < (n + 1)) ==>
           ((asize (lookup c j)) = (n + 2)))) /\
(k > 0) /\
(k = x)
<<
 fsum := sigma k n (lookup factors);
 c := c'.
    (contains_batch_costi c' (lookup times)
                         (lookup factors) n s (k - 1)) /\
    ((asize c') = (n + 1)) /\
    (!j. (j < (n + 1)) ==> ((asize (lookup c' j)) = (n + 2)));
 k := k - 1
>>
((k <= n) /\
(fsum = (sigma (SUC k) n (lookup factors))) /\
(contains_batch_costi c (lookup times)
                    (lookup factors) n s k) /\
((asize c) = (n + 1)) /\
(!j.  (j < (n + 1)) ==>
           ((asize (lookup c j)) = (n + 2)))) /\
(k < x)
```

The correctness conjecture that the body of the loop establishes the invariant at each step is trivial since the same predicate is used in both the nondeterministic assignment and the invariant. Therefore, this conjecture can be established easily. It is converted into a logical formula using the Sequence and Step commands of the Correctness extension and these formulas are proved using the Simplify option of the same extension.

```
(((asize c) = (n + 1)) /\
```

```
(!j. (j < (n + 1)) ==>
         ((asize (lookup c j)) = (n + 2)))) ==>
((n <= n) /\
(0 = (sigma (SUC n) n (lookup factors))) /\
(contains_batch_costi c (lookup times)
                    (lookup factors) n s n) /\
((asize c) = (n + 1)) /\
(!j.  (j < (n + 1)) ==>
           ((asize (lookup c j)) = (n + 2))))
```

The conjecture $pre \rightarrow inv$ can be established by simplifying the right hand side of the implication, expanding the definition of the sigma operator which returns 0 if the upper bound is less than the lower one. Finally expanding the definition of `contains_batch_costi` also reduces this formula trivially to true, since there is no row after row `n`.

```
(~ (k > 0)) /\
((k <= n) /\
 (contains_batch_costi c (lookup times)
                      (lookup factors) n s k) /\
 ((asize c) = (n + 1)) /\
 (!j.  (j < (n + 1)) ==>
            ((asize (lookup c j)) = (n + 2)))) ==>
((sigma (SUC k) n (lookup factors)) =
                    (sigma 1 n (lookup factors))) /\
(contains_batch_cost c (lookup times) (lookup factors) n s)
```

The conjecture $(\neg Guard \wedge invariant) \Rightarrow postcondition$ is satisfied since all the rows of `c` are completed when `k` reaches 0^{th} row, which is dummy. When its last argument is 0, the predicate `contains_batch_costi` becomes equal to `contains_batch_cost`.

Before going on to the implementation of the filling in of a single row, the value to be assigned to `fsum` must be computed by the algorithm. Although the assignment is a deterministic one, the value to be assigned is still calculated by a mathematical function that does not have a direct counterpart in our programming language. Notice that by the invariant, the value of `fsum` at the beginning of each loop execution equals to `(sigma (k + 1) n (lookup factors))`. To use this fact when computing the new value of fsum, `(sigma k n (lookup factors))` is rewritten as `(lookup factors k) + (sigma (k + 1) n (lookup factors))` using the theorem *sigma2_THM*. After this, the second term of the addition is replaced by the previous value of `fsum`.

The second loop is introduced as a refinement of the assignment to `c` in the first loop. This inner loop implements the assignment of a set of values to one row of `c`

by making an individual assignment to a single entry of the row with each execution. Similar to the way `k` and `fsum` were handled in the outer loop, a loop variable `m` and `tsum` are introduced and initialized to `k+1` and 0 respectively.

```
guard: m <= (n+1)
body: tsum := (sigma k (m-1) (lookup times));
      c := update c k (update (lookup c k) m
            (batchcost k m (lookup times) (lookup factors) n s));
      m := (m+1)
invariant:
     (k > 0) /\ (k <= n) /\
     (fsum= (sigma k n (lookup factors)))  /\
     (m> k) /\ (m <= (n+2))  /\
     (tsum= (sigma k ((m-1)-1) (lookup times)))  /\
     (contains_batch_costi c (lookup times)
            (lookup factors) n s k) /\
     (contains_batch_costrowj c (lookup times)
            (lookup factors) n s k (m-1)) /\
     (!j. (j < (n + 1)) ==>
              ((asize (lookup c j)) = (n + 2))) /\
     ((asize c) = (n + 1))
variant: (n+2) - m
```

In the body, the k^{th} row of `c` is accessed, and after its m^{th} element is updated, the resulting row replaces the old one. The function batchcost is equation (III.4) expressed in terms of HOL functions. This time, the invariant states that the entries of the k^{th} row up to column `m` contain the correct values. When `m` reaches `n+1`, the whole row has been filled.

The invariant contains a new predicate `contains_batch_costrowj` which is an even smaller ranged version of `contains_batch_cost`. It checks the positions of a single row, in this case `k`, from `k+1` upto and including its last argument, in this case `(m-1)`, for storing the cost of corresponding batches.

The invariant also contains information about the range of the outer loop variable `k` in addition to the range of `m`, the rows that have been filled previously, the size of `c` and the value of `fsum` which is not changed in the loop. While some of this information may seem redundant and not directly related to the loop, actually all are necessary for the proofs of the conjectures that are produced by the loop.

Since the correctness conjecture that appears below exceeds the sizes of the Tcl structures, the focusing mechanism of the Calculator fails. In order to advance, manual focusing is done by entering a HOL command and refreshing the display using Show

Stack.

```
(((k > 0) /\ (k <= n)) /\
 (fsum = (sigma k n (lookup factors))) /\
 ((m > k) /\ (m <= (n + 2))) /\
 (tsum = (sigma k ((m - 1) - 1) (lookup times))) /\
 (contains_batch_costi c (lookup times)
                    (lookup factors) n s k) /\
 (contains_batch_costrowj c (lookup times)
                    (lookup factors) n s k (m - 1)) /\
 (!j.  (j < (n + 1)) ==>
           ((asize (lookup c j)) = (n + 2))) /\
 ((asize c) = (n + 1))) /\
(m <= (n + 1)) /\
(((n + 2) - m) = x)
<<
   tsum := sigma k (m - 1) (lookup times);
   c := update c k (update (lookup c k) m
         (batchcost k m (lookup times) (lookup factors) n s));
   m := m + 1
 >>
 (((k > 0) /\ (k <= n)) /\
  (fsum = (sigma k n (lookup factors))) /\
  ((m > k) /\ (m <= (n + 2))) /\
  (tsum = (sigma k ((m - 1) - 1) (lookup times))) /\
  (contains_batch_costi c (lookup times)
                      (lookup factors) n s k) /\
  (contains_batch_costrowj c (lookup times)
                      (lookup factors) n s k (m - 1)) /\
  (!j.  (j < (n + 1)) ==>
             ((asize (lookup c j)) = (n + 2))) /\
  ((asize c) = (n + 1))) /\
 (((n + 2) - m) < x)
```

The challenging part of the proof of this correctness conjecture results from the assignment to `c`. Because the whole 2D array is captured in one value of type `((num) array) array`, the change of even a single element means the change of the whole value of the array, and we need the theorems of the array theory in order to relate these two values. First of all, it is to be proven that the rows `k+1` through `n` of the array still contain the same values. For this purpose, the theorem *update_lookup2* is used. This theorem states that the entries of the new array, excluding the one that has been updated, are equal to their previous values. The same theorem is used for proving that the previous range of the `contains_batch_costrowj` is preserved as elements `(k+1)` to `m-1` of the $\texttt{k}^{th}$ row.

The extension in the range of this predicate,i.e. the case of the newly updated element, is handled separately. The theorem *update_lookup1* is used to prove that when lookup function is applied to the updated element, the new value is returned. Finally, that the size of the updated array c is the same as the original one is proved using the theorem *update_asize*.

```
(n >= k) /\
(contains_batch_costi c (lookup times)
                    (lookup factors) n s k) /\
((asize c) = (n + 1)) /\
(!j.  (j < (n + 1)) ==>
           ((asize (lookup c j)) = (n + 2))) /\
(k > 0) /\
(fsum = (sigma k n (lookup factors))) ==>
((k > 0) /\ (k <= n)) /\
(fsum = (sigma k n (lookup factors))) /\
(((k + 1) > k) /\ ((k + 1) <= (n + 2))) /\
(0 = (sigma k (((k + 1) - 1) - 1) (lookup times))) /\
(contains_batch_costi c (lookup times)
                    (lookup factors) n s k) /\
(contains_batch_costrowj c (lookup times)
                    (lookup factors) n s k ((k + 1) - 1)) /\
(!j.  (j < (n + 1)) ==>
           ((asize (lookup c j)) = (n + 2))) /\
((asize c) = (n + 1))
```

The only new conjunct in the invariant is the one with the predicate contains_batch_costrowj. This reduces trivially to true because the lower and upper bound are the same, there exists no element for the predicate to apply.

```
(~ (m <= (n + 1))) /\
(((k > 0) /\ (k <= n)) /\
 (fsum = (sigma k n (lookup factors))) /\
 ((m > k) /\ (m <= (n + 2))) /\
 (contains_batch_costi c (lookup times)
                     (lookup factors) n s k) /\
 (contains_batch_costrowj c (lookup times)
                     (lookup factors) n s k (m - 1)) /\
 (!j.  (j < (n + 1)) ==>
            ((asize (lookup c j)) = (n + 2))) /\
 ((asize c) = (n + 1))) ==>
(contains_batch_costi c (lookup times)
                    (lookup factors) n s (k - 1)) /\
((asize c) = (n + 1)) /\
(!j.  (j < (n + 1)) ==>
           ((asize (lookup c j)) = (n + 2)))
```

At the completion of the execution of the loop, `m` equals `n+2`, which can be inferred from the information the negated guard and invariant provide together on the left hand side. Using this substitution, `contains_batch_costrowj` tells us that all the elements between `k` and `n+1`, including this last entry, now contain the values we need. In the proof, after the substitution of the value of `m`, the definitions of the predicates that talk about `c` on both sides of the implication are expanded. Then the terms that result from the expansion of `contains_batch_costi` that occurs on the right hand side is brought to focus. The case for rows numbered greater than `k` are already contained in the expansion of the same predicate on the left hand side. For the k^{th} row, the information that comes from `contains_batch_costrowj` that the whole row has been filled in is used.

The algorithm that has been driven for the first stage is shown below.

```
var times:num array,factors:num array,s:num,n:num,c:num array array.
 |[var fsum:num | T .
   |[var k:num | T .
     fsum := 0;
     k := n;
     do k > 0 ->
           fsum := (lookup factors k) + fsum;
           |[var tsum:num | T .
             |[var m:num | T .
               tsum := 0;
               m := k + 1;
               do m <= (n + 1) ->
                     tsum := (lookup times (m - 1)) + tsum;
                     c :=
                        update c k (update (lookup c k) m
                         (batchcost k m (lookup times)
                          (lookup factors) n s));
                     m := m + 1
               od
             ]|
           ]|;
           k := k - 1
     od;
   ]|
 ]|
```

Figure III.3: First Part of the Algorithm

III.4 Computing the Start of Next Batch

The precondition of the second stage of the algorithm assumes the 2D array `c`, which is the product of the first stage. Two new arrays are declared: `next` and `costn`, which correspond to *next* and F in the algorithm mentioned in III.1. (The name F could not be used since the keyword F is reserved to represent the logical false in HOL)

```
program computeXtoNcost
var times:(num) array; factors:(num) array; s:num; n:num;
c:((num) array) array; next:(num) array; costn:(num) array.
{
(n >= 1) /\
(contains_batch_cost c (lookup times) (lookup factors) n s) /\
((asize next) = (n+1)) /\
((asize costn) = (n+2))
};
next,costn := next', costn'.
(?p. (contains_partition next' 1 n p) /\
 (sub_min_cost_p  (lookup times) (lookup factors) 1 n s p) /\
( (lookup costn' 1) =
    (overallcost  (lookup times) (lookup factors) n s p) )
)
```

The postcondition is comprised of a nondeterministic assignment to the two arrays `next` and `costn`. The quantification in this condition signifies the existence of a partition, in association to these arrays, satisfying the property of being minimum-costed. After the execution of the algorithm, the 1^{st} element of the array `costn` should contain the cost of the partition `p` with the desired properties. This value is guaranteed to be minimal by the `sub_min_cost_p` predicate.

Our algorithm computes minimum-costed partitions for not just the first job but for all the jobs in the queue. So the next step is to refine the nondeterministic assignment as:

```
next,costn := next', costn'.
(!x. ((1 <= x) /\ (x <= n)) ==>
(?p. (contains_partition next' x n p) /\
 (sub_min_cost_p  (lookup times)
               (lookup factors) x n s p) /\
((lookup costn' x) =
     (overallcost  (lookup times) (lookup factors) n s p))
))
```

The predicate `contains_partition` checks that for each job `x` in the queue, a partition starting from `x` is embedded in the final value of `next`. Each such partition is

also required to have minimum cost amongst the others that contain the jobs `x` through `n`, by the `sub_min_cost_p` predicate. This predicate selects upto the time and cost factor functions, minimal costed partitions that start with a certain job. Meanwhile, all the properties of being a partition are also checked. In addition to starting with `x`, the elements of the partition should be strictly increasing and the last job of the partition is to be `n+1`. The function `overallcost` merely calculates the cost of one partition, adding up the cost of each batch. Hence, the final value of `costn` will contain the minimal cost of executing jobs from `x` to `n` in the position `x` and in the same position of array `next` the start of the second batch of such a partition will be stored.

Next an assignment is added prior to the one in the specification. 0 is stored in the $(\texttt{n}+1)^{st}$ position of the `costn` array, as the cost of reaching from job `(n+1)` to again job `(n+1)`. This represents the cost of an empty partition which is trivially contained in the `next` array.

For the example mentioned in section III.1, the arrays in figure III.4 would be produced by the algorithm. The corresponding partition that includes all the jobs is shown in figure III.5.

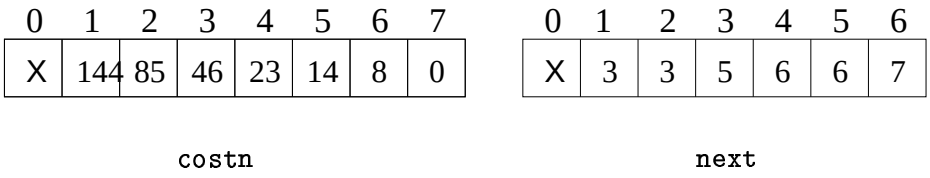

Figure III.4: Final values of `costn` and `next` arrays

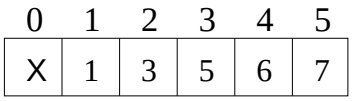

Figure III.5: Final partition

The following loop implements the above specification using the following dynamic programming idea: The cost of a partition from `k` to `n+1` is formed using a known partition from some `kk` to `n+1` , where `kk` $>$ `k`. New partitions are created by adding a new job to the head of an old one and so the cost of a new partition is calculated from the cost of an old one by the function `computecost` defined as `(\(kk:num). (batchcost`

j kk times factors n s) + (mincostf kk)) which corresponds to equation (III.5).

Before the introduction of the loop, a loop variable k is introduced, and initialized to n. This is because n is the start of the first partition that has to be constructed.

```
guard:  k > 0
body: next,costn := next', costn'.
      (min_costed_next (lookup costn)  times factors k n s
          (lookup next' k)) /\
      ((asize next') = (asize next)) /\
      (!i. ((k < i) /\ (i < (asize next'))) ==>
           ((lookup next' i) = (lookup next i))) /\
      (min_cost (lookup costn) times factors k n s
          (lookup costn' k)) /\
      ((asize costn') = (asize costn)) /\
      (!i. ((k < i) /\ (i < (asize costn'))) ==>
           ((lookup costn' i) = (lookup costn i)));
      k:= k - 1
invariant:
     (contains_batch_cost c
                 (lookup times) (lookup factors) n s) /\
     (k <= n) /\
     (!x. (((SUC k) <= x) /\\ (x <= n)) ==>
          (?p. (contains_partition next x n p) /\
               (sub_min_cost_p (lookup times)
                           (lookup factors) x n s p) /\
               ((lookup costn x) =
                    (overallcost (lookup times)
                           (lookup factors) n s p)))) /\
     ((lookup costn (n+1)) = 0) /\
     ((asize next) = (n+1)) /\
     ((asize costn) = (n+2))
 variant: k
```

With each execution of the body of the first loop, the two arrays next and costn will change so that their k^{th} entries satisfy the predicates min_costed_next and min_cost, respectively. The k^{th} entry of costn will hold the minimum value that the computecost function can take in the range (k < kk) / (kk <= n+1), and next[k] will hold the index kk where this minimum value is obtained. Here mincostf is intended as a function that yields for kk in the range (k < kk) / (kk <= n+1), the overall costs of the minimum costed partitions from kk to n. In fact, the array costn contains these costs.

```
(contains_batch_cost c0
       (lookup times0) (lookup factors0) n0 s0) /\
```

```
((asize next0) = (n0 + 1)) /\
((asize costn0) = (n0 + 2)) /\
((lookup costn0 (SUC n0)) = 0) ==>
(n0 <= n0) /\
(!x.
   (((SUC n0) <= x) /\ (x <= n0)) ==>
      (?p.
        (contains_partition next0 x n0 p) /\
        (sub_min_cost_p (lookup times0)
                  (lookup factors0) x n0 s0 p) /\
        ((lookup costn0 x) =
          (overallcost (lookup times0)
                     (lookup factors0) n0 s0 p))
    )) /\
((lookup costn0 (n0 + 1)) = 0) /\
((asize next0) = (n0 + 1)) /\
((asize costn0) = (n0 + 2))
```

The precondition implies the invariant because before any execution of the loop, no partitions which contain jobs are expected to be embedded in the `next` array.

```
(~ (k0 > 0)) /\
(contains_batch_cost c0 (lookup times0)
          (lookup factors0) n0 s0) /\
(k0 <= n0) /\
(!x.
   (((SUC k0) <= x) /\ (x <= n0)) ==>
       (?p.
        (contains_partition next0 x n0 p) /\
        (sub_min_cost_p (lookup times0)
           (lookup factors0) x n0 s0 p) /\
        ((lookup costn0 x) =
           (overallcost (lookup times0)
                 (lookup factors0) n0 s0 p)))) /\
((lookup costn0 (n0 + 1)) = 0)) /\
((asize next0) = (n0 + 1)) /\
((asize costn0) = (n0 + 2)) ==>
   (!x.
     ((1 <= x) /\ (x <= n0)) ==>
     (?p.
       (contains_partition next0 x n0 p) /\
       (sub_min_cost_p (lookup times0)
                 (lookup factors0) x n0 s0 p) /\
       ((lookup costn0 x) =
             (overallcost (lookup times0)
                 (lookup factors0) n0 s0 p))))
```

When the guard becomes false, `k` is 0 and so partitions starting with the jobs `n0`

through 1 has been constructed by the loop. And the costs of these minimally costed arrays have been stored in `costn`.

Instead of the correctness conjecture that is produced by this loop, the theorem which is the key to its proof will be given here. This theorem implicitly contains the proof of the shortest-path algorithm and is the most challenging proof in this study.

If we consider the correctness conjecture, it will ask for a new minimum costed partition at the end of each loop containing the jobs `k` through `n`. In the algorithm, this is accomplished by the creation of a new partition from an old one (one that is already embedded in the `next` array). The choice of this old partition is done according to the `computecost` function that has to be minimized and the new partition is formed by just adding the currently considered job to the beginning of the old partition. These observations lead to the formation of the theorem below.

```
(!(k:num).  (!(times:(num) array).  (!(factors:(num) array).
(!(s:num). (!(n:num). (!(next:(num) array).
(!(costn:(num) array).
 (min_costed_next (lookup costn) (lookup times)
             (lookup factors) k n s (lookup next k)) /\
 (min_cost (lookup costn) (lookup times)
             (lookup factors) k n s (lookup costn k)) /\
 (contains_partition next (lookup next k) n p) /\
 (sub_min_cost_p (lookup times)
             (lookup factors) (lookup next k) n s p) /\
 ((lookup costn (lookup next k)) =
             (overallcost (lookup times)
                          (lookup factors) n s p)) /\
 (!x'.  (((SUC k) <= x') /\ (x' <= (n+1))) ==>
    (?p.  (contains_partition next x' n p) /\
          ((sub_min_cost_p (lookup times)
                           (lookup factors) x' n s p) /\
          ((lookup costn x') =
                  (overallcost (lookup times)
                               (lookup factors) n s p))))) /\
 (k <= n) /\ (k > 0) /\ (x = k) /\
 ((lookup costn (n + 1)) = 0) /\
 ((asize next) = (n + 1)) /\
 ((asize costn) = (n + 2))
==>
 (?p'.  (contains_partition next k n p') /\
        (sub_min_cost_p (lookup times)
                    (lookup factors) k n s p') /\
        ((lookup costn k) =
                  (overallcost (lookup times)
                               (lookup factors) n s p')) /\
```

```
(p' = (addp k p)))))))))))
```

Consider the point after the nondeterministic assignment in the loop body. The $\mathtt{k}^{th}$ positions of the two arrays now satisfy the predicates `min_costed_next` and `min_cost` as is expressed in the first two conjuncts of the left hand side. Since the job that is stored in position `k` of `next` has index larger than k, by the invariant, we can safely assume a corresponding minimal costed partition. The overall cost of this partition would be stored in position `lookup next k` of the `costn` array. Other partitions that belong to jobs of larger indexes can also be assumed relying on the invariant, except the $(\mathtt{n}+1)^{st}$ one.

A partition exists for this job also. But it is not produced by the loop. Such a partition would have size 2, consisting of the 0th element that is not used and the first and last element would be `(n + 1)`. We call it the empty partition since it contains no jobs. This partition is necessary solely for the creation of a partition with one element. Since it contains no batches it is trivially contained in the `next` array and its cost is stored in the $(\mathtt{n}+1)^{st}$ position of `costn` as is required of other partitions.

The right hand of the implication is clear except the last conjunct. This conjunct serve to relate the partition that runs from `lookup next k` to `n` to the new partition that starts with `k`. The function `addp` takes a job number and a partition as arguments and adds a batch starting from the given job to the beginning of this partition. Notice that for the result to be a valid partition, it is necessary and sufficient for the given job to be numbered smaller than the first job of the partition.

In order to prove this theorem, a constructive proof was done. The new partition has been created by applying the `addp` function to an old partition, and it is proven that this new partition satisfies the necessary properties of being embedded in `next` array, being a valid partition and having minimum cost amongst all other valid partitions that begin with the job `k`.

The property of being embedded in `next` array requires a rearrangement of indexes, since all the positions of the heads of batches in the old partition has been shifted by one position. The fact that `k > (lookup next k)`, combined with the properties of the old partition `p` enable us to prove that the new partition `p'` is a valid one.

The minimality proof which is the heart of the whole algorithm is related with the second conjunct. Like most minimality proofs, a partition `q` with a smaller cost than

p is supposed to exist. q also starts from job k. Since we know it contains at least two elements, (k and n+1), q can not be the empty partition and so it can be imagined as a composite partition like p.

At this point, the counterpart of addp, the subp function is introduced. This function erases the first batch from its argument (a partition) as long as this argument is not the empty partition.

So q is considered as being equal to (addp (lookup q 1) (subp q)) using the theorem *addp_subp*. (subp q) is a partition in its own right, which is captured in the theorem *subp_vsp*. The cost of q can be rewritten using the theorem addp_overallcost which computes the cost of a composite partition from its parts.

The overall cost of q is given by:

```
((overallcost (lookup times) (lookup factors) n s (subp q)) +
(batchcost k (lookup (subp q) 1) (lookup times) (lookup factors) n s))
```

Suppose (subp q) is a minimally costed partition starting from job (lookup (subp q) 1) or else its minimality would be in question since it would be replaced by a partition starting with the same job but having minimum cost. So its cost is stored in position (lookup (subp q) 1) of costn array. This allows us to rewrite the above sum as:

```
(lookup costn (lookup (subp q) 1)) +
(batchcost k (lookup (subp q) 1) (lookup times) (lookup factors) n s))
```

Because of the partition properties, we know (lookup (subp q) 1) is in the range (k, (n + 1)] and so must have been considered for the k^{th} position of next array. The fact that it has not been chosen means that the above sum yields a higher one than the same sum for p. So we have reached a contradiction.

Finally the conjunct about costn is left. The predicate min_value requires the existence of an element in the domain that produces the minimum cost but does not specify which element it is, in the meanwhile it is known by the min_costed_next predicate that (lookup next k) is one such minimizing element. Through the observation that minima is unique in a certain range (theorem *unique1*) , the cost yielded by the (lookup next k) and (lookup costn k) are found to be equal.

The nondeterministic assignment in the body of the loop is refined as an inner loop, which satisfies the predicates min_costed_next and min_cost by updating the corresponding k^{th} entries. Before any execution, the number of the first job of the next batch

is set as the next one in the queue, `(k+1)`, and the minimal cost of such a partition, where the first batch consists of the first job only, is assigned to `costn[k]`. Since the initial values (unprimed) of both arrays occur in the postcondition, immediate addition of these assignments would change the relation between the final and initial values set by the postcondition. In order to be able to add these assignments, the values of the arrays before the nondeterministic assignment is captured in two constants named `NEXT` and `COSTN`, using the Use Specification Constant option. The equalities `nextn = NEXT` and `costn = COSTN` are then added to the assertion. Using these equalities, the unprimed occurrences of these variables are replaced by constants in the nondeterministic assignment. Finally, the following leading assignment is added:

```
m,costn,next:=
k+2,
(update costn k
   ((batchcost k (k+1) (lookup times) (lookup factors) n s) +
    (lookup costn (k+1)))),
(update next k (k+1))
```

The inner loop introduced below scans the upper part (from index `k+2` to `n+1`) of the job queue for the start of the next batch that produces the minimum costed partition. Tracking the inner loop on our example where for `k` = 3 is shown in III.6

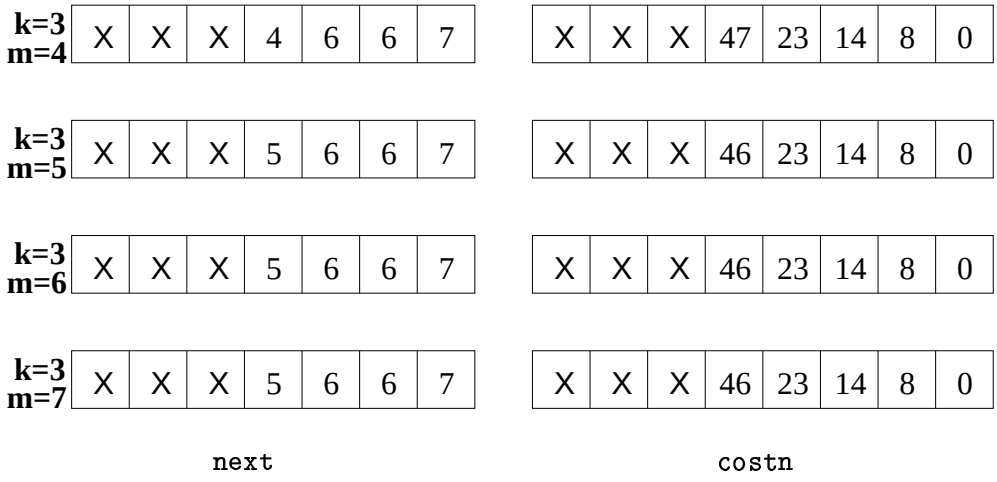

Figure III.6: The execution of the inner loop

```
guard: m <= (n+1)
body: costn,next:=
      update costn k
```

```
            (minvalue2
               (computecost costn times factors n s k)
                                 (lookup next k) m ),
      update next k
            (minimizes2
               (computecost costn times factors n s k)
                                 (lookup next k) m );
      m := m+1
invariant: (m <= (n+2)) /\ (m > (k+1)) /\
      (contains_batch_cost c
         (lookup times) (lookup factors) n s) /\
      (minvalue  (computecost costn times factors n s k)
         (\(kk:num). (kk>k) /\ (kk <=(m-1))) (lookup costn k)) /\
      (minimizes (computecost costn times factors n s k)
         (\(kk:num). (kk>k) /\ (kk <=(m-1))) (lookup next k)) /\
      ((asize next) = (asize NEXT)) /\
      (!i. ((k < i) /\ (i < (asize next))) ==>
              ((lookup next i) = (lookup NEXT i))) /\
      ((asize costn) = (asize COSTN)) /\
      (!i. ((k < i) /\ (i < (asize costn))) ==>
              ((lookup costn i) = (lookup COSTN i))) /\
      ((lookup costn (n+1)) = 0) /\
      ((asize next) = (n+1)) /\
      ((asize costn) = (n+2)) /\
      (k <= n) /\ (k > 0)
variant: (n+2) - m
```

The invariant merely extends the range of jobs that have been considered as the start of next batch, one job at a time. The functions `minvalue2` and `minimizes2` consider two jobs for the starter of the next batch. They both evaluate the `computecost` function at the two points. The smaller of these evaluated values is returned by `minvalue2`, while `minimizes2` returns the job which has yielded this smaller value. One interesting fact in the invariant is the preservation of the values in the indexes higher than the one considered `k < i`. Since only one position (`k`) is considered in the inner loop, no other positions of the two arrays are altered. But it is enough to state that the upper indexes of the initial values (`NEXT, COSTN`) have not changed.

The correctness conjecture produced by this loop is proved using the relationship between `minvalue-minvalue2` and `minimizes-minimizes2`. While the former members of these pairs apply to a range, the latter compare only two arguments. The minimum values of the range are computed by comparing a new element with the minimum of the group, and hence enlarging the group which has been considered.

The body of the second loop is further refined taking into consideration the follow-

ing points. First, after the definitions of `minvalue2`, `minimizes2` and `computecost` are expanded in the above algorithm, it is seen that several computations may be omitted. For example, `batchcost j kk times factors n s` is rewritten as `c`[`j`][`kk`] in the definition of `computecost` function using the information about `c` that has been propagated in the assertion through the two loops. Second, when comparing the costs of the already stored index in `next` and the new candidate `m`, the result of the previous cost computation stored in `costn` can be used. Thus avoiding duplicate cost computations.

The definitions of the former two predicates as HOL conditionals naturally bring about the introduction of the conditonal contruct. The following guard is entered to Cond Introduction.

```
((lookup (lookup c k) m) + (lookup costn m)) <
((lookup (lookup c k) (lookup next k)) +
                   (lookup costn (lookup next k)))}
```

Finally, in case the minimals have not changed, the update of `k`th entries of `next` and `costn` by their previous values can be refined to the `skip` statement using *update_id* theorem of the array theory.

The final algorithm appears in figure III.7.

We have proceeded by introducing two loops, one in the body of the other. The outer loop determines the partition for each job in the queue, while the inner loop chooses the start of next batch for a particular job by scanning the upper part of the array. A conditional construct is introduced in the body of the inner loop to compare two values and act according to this comparison.

```
var times:num array,factors:num array,s:num,n:num,
c:num array array, next:num array,costn:num array.
{(n >= 1) /\
 (contains_batch_cost c (lookup times)
                          (lookup factors) n s) /\
 ((asize next) = (n+1)) /\
 (((asize costn) = (n+2))};
 costn:= update costn (SUC n) 0;
 |[var k:num | T .
    k := n;
    do k > 0 ->
    |[var m:num | T .
       m,next,costn :=
           k + 2, update next k (k + 1),
           update costn k
           (lookup (lookup c k) (k + 1));
         do m <= (SUC n) ->
            if (((lookup (lookup c k) m) +
                          (lookup costn m))
                 < (lookup costn k))
               then
                    costn :=
                       update costn k
                       ((lookup (lookup c k) m) +
                                    (lookup costn m));
                    next := update next k m
               else
                    skip
            fi;
            m := m + 1
          od
    ]|;
     k := k - 1
     od
 ]|
```

Figure III.7: The Second Part of the Algorithm

CHAPTER IV

CONCLUSION

IV.1 Discussion

In this study, Refinement, Correctness and General Logic extensions were used. Most of the menu options of these extensions have been applied. While the backbone of the system remains intact as long as the size of the expression in focus is not very large, various focusing and highlighting errors occur. For example, sometimes the tool can not correctly determine the assumption that is currently selected and the user is forced to do this manually on the "HOL side" by extracting the necessary information from the stack information kept by the Calculator.

Another problem is the error messages given by Tcl/Tk because of various synchronization problems between the interface and HOL. Most of these messages can be safely ignored. However, since the steps that cause errors are not recorded, such derivations can not be restored. What is more, from time to time as a result of synchronization problems, the Focus window gets distorted. In this case, the Focus window should be refreshed manually using the Show Stack option. The problem is a persistent one, the need to manually refresh recurs at each step thereafter, and hence the derivation slows down.

But as the proof obligations get larger and more complicated, the problems get too severe to ignore. Tcl can not handle the necessary information and so the focusing mechanism fails altogether. Fortunately, the information is kept intact in HOL structures, and the user can continue the derivation by manually focusing on some subterm.

This fixes the focusing mechanism by reducing the size of the information handled by the interface. In order to manually focus on a subterm, the user has to know about the complex encoding used to address terms in the focus by the Calculator. This type of problem may be prevented to some extent by dealing with proof obligations as they are produced instead of accumulating them, and by expanding predicate definitions only at points of need. These precautions would help control the amount of information load on the interface.

Transformational reasoning is very suitable for making refinements and so keep the derivation phase tidy and clear. The representation of context information in the middle window which appears when one focuses on a subterm immediately after an assertion, especially enables a clear view of the situation.

However, the calculator is weak in the support it provides for proving obligations for the refinements. The greater portion of the time used to make derivations is spent on these proofs while actually they are mostly trivial from the mathematical standpoint. The process requires the user to have a strong background on HOL, in spite of the options provided by the General Logic extension. Particular HOL theorem names must be known and usually knowledge of functions to modify these theorems are also needed.

Many improvements are possible to overcome this shortcoming. For example, in the proof statements that result from correctness conjectures, two values occur for each variable (The initial value of variable `a` is represented by `a0` and the final value by `a'`). If the value of a variable has not changed after the execution of the program segment in question, equations of the form `(a' = a0)` appear. the difference in the representation of the two values and the redundant equations serve only to decrease readability. A simplification of the proof statement before it is displayed to the user is desirable.

Some of the power of the Simplify command could be distributed to other menu choices that transform the focus using theorems (e.g. Transform with Theorem, Conditional Rewrite) which currently require exact pattern matching.

Another major drawback of the tool is the insufficient information provided by the error messages and the lack of detail in the manual [7]. The only distinguishable errors are of syntax and even these do not guide the user on what should be done. The manual has on average three sentences for each menu option which forces the user to a guessing-trial-error loop. Various information about each option, like the arguments it expects, conditions for a successful application and its result, should be documented

in detail.

As for programming features, there is more to do in issues of abstraction. It is not possible to achieve data abstraction using the Calculator interface. As for functional abstraction, there are problems about blocks and procedures. It is not possible to introduce multiple variables at once and use them in loops. Similarly, problems occur when introducing loops in procedures. This has ruled out the possibility of a recursive implementation for the problem.

Using the advantages of transformational reasoning so well, the the use of the Calculator in practical applications is not a far fetched idea. This study stregthens the idea that the theory is an effective one. However, stronger support for domain specific theories like graph theory, and programming constructs like blocks and procedures remains crucial.

IV.2 Future Work

It is possible to further refine this study in various aspects. The lack of a "partition" theory necessitated the modeling of partitions as `num` arrays. Although partitions are comprised of a sequence of numbers, they have additional constraints like being strictly increasing, starting with a given number (the first job) and ending with *N+1*, where *N* is the number of jobs to be partitioned. The semantics of the partition would also suggest referring to the elements with batch numbers, i.e. starting from 1, whereas the elements of the array are accessed starting from 0. Due to these additional properties of partitions, implementing them as a datatype would be desirable.

The datatype would be complete with functions that solely operate on partitions: `addp` and `subp`. Basic theorems like the preservation of partition properties with respect to these functions would also be included in the theory.

The "partition" datatype would clarify the representation, and the theorems included about functions would simplify the proof of the minimality theorem.

For the correctness conjecture that results from the introduction of the first loop of the second stage includes the implicit proof of the shortest path algorithm. A formulation of the proof of this conjecture to use the shortest path proof as a theorem would make the derivation more elegant. This type of reuse is to be the key to the derivation of algorithms for problems that can be reduced to another.

This study aims to evaluate the applicability of the approach and the performance of the tool in deriving dynamic algorithms. The derivation of the quadratic algorithm may be followed by the linear one mentioned in [1]. Derivation of the quadratic algorithm has already pointed the major shortcomings of the calculator that need to be remedied in future releases, as well as showing the methodology to be effective. The derivation of the linear algorithm would enable an evaluation of deriving algorithms of different characteristics.

REFERENCES

[1] S. Albers, P. Brucker *The complexity of one-machine batching problems*, Discrete Applied Mathematics 47,pp. 87-107, 1993

[2] M. J. C. Gordon, T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic*. Cambridge University Press, 1993

[3] R. J. Back. *On correct refinement of programs*, Journal of Computer System Sciences, 23(1):49-68, Feb 1981

[4] R. J. Back. *On the correctness of refinement in program development*, PhD Dissertation, Department of Computer Science, University of Helsinki, 1978

[5] R.J. Back, J. von Wright *Refinement Calculus: A Systematic Introduction*. Springer-Verlag. 1998

[6] M.J. Butler, J. Grundy, T. Langbacka, R. Ruksenas, J. von Wright. *The Refinement Calculator: Proof Support for Program Refinement* Proceedings of Formal Methods Pacific '97, Wellington, New Zealand, Springer-Verlag, July 1997

[7] M.J. Butler, T. Langbacka, R. Ruksenas, J. von Wright The Refinement Calculator Tutorial and Manual. Available upon request from the authors, 2003

[8] E.W. Dijkstra *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice Hall. 1976

[9] L. Laibinis *Mechanised Formal Reasoning About Modular Programs*, PhD Dissertation, TUCS, Finland, April 2000

[10] T. Langbacka, R. Ruksenas, J. von Wright *TkWinHOL: A tool for doing window inference in HOL* Higher Order Logic Theorem Proving and Its Applications, LNCS 971, Springer-Verlag, 1995

[11] C.C. Morgan *Programming from Specifications (2nd Edition)*. Prentice Hall, 1994

[12] R. J. Back. *Refinement Calculus, part II: Parallel and reactive programs* REX Workshop for Refinement of Distributed Systems, LNCS 430, Springer-Verlag, 1989.

[13] R. J. Back, K. Sere *Stepwise refinement of action systems* The Mathematics of Program Construction, LNCS 375, pg. 17-30, Springer-Verlag, 1989.

[14] M. Utting, K. Robinson *Modular reasoning in an object-oriented refinement calculus* In R.Bird, C.C. Morgan and J. Woodcock (Eds.), Mathematics of Program Construction, LNCS 669, 334-367, Springer-Verlag, 1993.

[15] C.C. Morgan *Data Refinement by miracles* Information Processing Letters, 26:243-246, 1988.

[16] P.H. Gardiner, C.C. Morgan *Data refinement of predicate transformers* Theoretical Computer Science, 87(1):143-162, 1991.

[17] C.C. Morgan, A.K.McIver, K. Seidel *Probabilistic predicate transformers* ACM Transactions on Programming Languages and Systems, 18(3):325-353, 1996.

APPENDIX A

NOTATION INDEX

Table A.1: Notation Index

Math Notation	HOL Notation	Explanation
$i-1$	`PRE(i)`	Predecessor Operator
$i+1$	`SUC(i)`	Successor Operator
$\wedge$	`/\`	Conjunction
$\vee$	`\/`	Disjunction
$\rightarrow$	`==>`	Implication
$=$	`=`	Equality
$\forall t \in T.P(t)$	`! t : T . (P t)`	Universal Quantifier
$\exists t \in T.P(t)$	`? t : T . (P t)`	Existential Quantifier
if t then s else u	`t => s | u`	Conditional Term
λ	`\ t : T . (F t)`	Lambda abstraction

APPENDIX B

PREDICATES AND FUNCTIONS

B.1 Definitions of Array Theory

```
init_array (x:'a) s =
       ABS_array ((\(j:num). (j < s => x | (@y.T))),s)
```

init_array($'a \rightarrow (num \rightarrow (('a)array))$) returns an array of size `s`, all the elements in the range 0 to `s-1` are initialized to `x`. The indexes out of bounds may contain any element of type ('a).

B.2 User Defined Functions and Predicates

```
 addp (first:'a) (a:('a)array) = ABS_array
   ((\j:num.(j > 1) => lookup a (j-1) |
                    ((j=1) => first | lookup a 0)),(asize a+1))
```

addp($'a \rightarrow ((('a)array) \rightarrow (('a)array))$) returns an array which has `first` inserted in the first position of `a`. The elements of `a` in indexes 1 and higher have been shifted one position right. The value stored in position 0 is not changed. The size of the array returned is one more than the size of `a`.

```
 batchcost fjob nextfjob (times:num->num) (factors:num->num) (n:num)
(s:num)=
    (sigma fjob n factors) * (s + sigma fjob (nextfjob-1) times)
```

batchcost($num \rightarrow (num \rightarrow ((num \rightarrow num) \rightarrow ((num \rightarrow num) \rightarrow (num \rightarrow (num \rightarrow num))))))$ returns the cost of a batch which contains the jobs `fjob` through `(nextfjob-1)` (`nextfjob` is the first job of the next batch). The execution times and

cost factors of the n jobs in the queue can be found using the functions times and factors. It is III.4 in HOL notation.

```
 computecost (mincostf:num array) (times:num array) (factors:num array)
(n:num) (s:num) (job:num)=
    \(nextjob:num).
       ((batchcost job nextjob (lookup times) (lookup factors) n s) +
                 (lookup mincostf nextjob))
```

computecost$((numarray) \rightarrow ((numarray) \rightarrow ((numarray) \rightarrow (num \rightarrow (num \rightarrow (num \rightarrow (num \rightarrow num)))))))$ returns the function that given a job numbered higher than job, it calculates the minimum cost of a partition consisting of jobs job through n, and that has nextjob as the start of the second batch. (It is assumed that minimum path costs are stored in the array mincostf for these higher numbered jobs.)

```
 contains_batch_cost (carray:(num array) array) (times:num->num)
(factors:num->num) (n:num) (s:num)=
     (! i j. ( (0 < i) /\ ( i < j ) /\ (j <= (n+1)) ) ==>
    (( lookup (lookup carray i) j) = (batchcost i j times factors n s))
   )
```

contains_batch_cost$((((num)array)array) \rightarrow ((num \rightarrow num) \rightarrow ((num \rightarrow num) \rightarrow (num \rightarrow (num \rightarrow bool)))))$ returns true if carray$[i][j]$ contains the cost of the batch containing the jobs i through $(j-1)$ when $i < j$. Since they correspond to job numbers, j can not be greater than n+1, and i should be greater than 0. In order for these positions to be defined, it is enough that carray is a matrix of size n x (n+1).

```
 contains_batch_costi (carray:(num array) array) (times:num->num)
(factors:num->num) (n:num) (s:num) (k:num)=
     (! i j. ( (k < i) /\ ( i < j ) /\ (j <= (n+1)) ) ==>
    (( lookup (lookup carray i) j) = (batchcost i j times factors n s))
   )
```

contains_batch_costi$((((num)array)array) \rightarrow ((num \rightarrow num) \rightarrow ((num \rightarrow num) \rightarrow (num \rightarrow (num \rightarrow (num \rightarrow bool))))))$ returns true if carray$[i][j]$ contains the cost of the batch containing the jobs i through $(j-1)$ where $i < j$, and k < i. Since it corresponds to a job number, j can not be greater than n+1. The predicate checks the rows numbered greater than k, of the matrix carray of size, at least, n x (n+1).

```
 contains_batch_costrowj (carray:(num array) array) (times:num->num)
(factors:
num->num) (n:num) (s:num) (k:num) (m:num)=
```

```
    (!j. ( (k < j) /\ (j <= m) ) ==>
       (( lookup (lookup carray k) j) = (batchcost k j times factors n s))
  )
```

contains_batch_costrowj $((((num)array)array) \rightarrow ((num \rightarrow num) \rightarrow ((num \rightarrow num) \rightarrow (num \rightarrow (num \rightarrow (num \rightarrow (num \rightarrow bool))))))$ returns true if `carray`[`k`][j] contains the cost of the batch containing the jobs `k` through $(j-1)$ where $j >$ `k` and $j <$ `m`. It checks the portion to the right of the diagonal of the row `k` of the matrix `carray`.

```
contains_partition (next:(num) array) (j:num) (n:num) (p:(num) array)=
       ((asize p) > 1) /\ (lookup p 1 = j) /\
       (!x. ((1 < x) /\ (x < (asize p)))
            ==>
               ( (lookup p x) = (lookup next (lookup p (x - 1))) ))
```

contains_partition$(((num)array) \rightarrow (num \rightarrow (num \rightarrow (((num)array) \rightarrow bool))))$ returns true if `next` contains the partition `p` encoded as is indicated by the last conjunct. In order to find the start of a batch of `p`, we access the `next` array using the first job of the previous batch. `p` contains the jobs `j` through `n` and so contains `j` in first postition and has size more than 1.

```
init_partition (n:num) =  (init_array (n+1) 2)
```

init_partition$(num \rightarrow ((num)array))$ returns the empty partition which contains no jobs. The array returned has `n+1` in its first position, and has size 2.

```
 min_costed_next (mincostf:num->num) (times:num->num) (factors:num->num)
 (j:num) (n:num) (s:num) (z:num)=
minimizes (\(k:num). (batchcost j k times factors n s) + (mincostf k))
(\(k:num). (k>j) /\ (k <=(n+1))) z
```

min_costed_next$((numarray) \rightarrow ((num \rightarrow num) \rightarrow ((num \rightarrow num) \rightarrow (num \rightarrow (num \rightarrow (num \rightarrow (num \rightarrow bool)))))))$ returns true if `z` yields the minimum value of the function `(\(k:num). (batchcost j k times factors n s) + (mincostf k))` in the range `(k>j) /\ (k <=(n+1))`.

```
 min_cost (mincostf:num->num) (times:num->num) (factors:num->num)
(j:num) (n:num) (s:num) (z:num) =
minvalue (\(k:num). (batchcost j k times factors n s) + (mincostf k))
(\(k:num). (k>j) /\ (k <=(n+1))) z
```

min_cost$((numarray) \rightarrow ((num \rightarrow num) \rightarrow ((num \rightarrow num) \rightarrow (num \rightarrow (num \rightarrow (num \rightarrow (num \rightarrow bool)))))))$ returns true if `z` is the minimum value that the function `(\(k:num). (batchcost j k times factors n s) + (mincostf k))` takes in the range `(k>j) /\ (k <=(n+1))`.

```
minvalue (f:'a->num) (s:'a->bool) (x:num)=
   ?(j:'a). ((s j) /\ (x = (f j)) /\ (!(k:'a). ((s k) ==> (x <= (f k)))))
```

minvalue $((\prime a \rightarrow num) \rightarrow ((\prime a \rightarrow bool) \rightarrow (num \rightarrow bool)))$ returns true if there is an element in the range determined by predicate `s` that yields the minimum value of function `f` in the same range and `x` is this value.

```
minimizes (f:'a->num) (s:'a->bool) (j:'a)=
     (s j) /\ (!(k:'a). (s k) ==> ((f j) <= (f k)))
```

minimizes $((\prime a \rightarrow num) \rightarrow ((\prime a \rightarrow bool) \rightarrow (\prime a \rightarrow bool)))$ returns true if `j` is the element that yields the minimum value of function `f` in the range determined by predicate `s`.

```
minvalue2  (f:'a->num) (a:'a) (b:'a) =
       (f(b) < f(a)  => f(b) | f(a))
```

minvalue2$((\prime a \rightarrow num) \rightarrow (\prime a \rightarrow (\prime a \rightarrow num)))$ returns `f(b)` if `f(b)` is less than `f(a)`, returns `f(a)` otherwise.

```
minimizes2 (f:'a->num) (a:'a) (b:'a)=
       (f(b) < f(a)  => b | a)
```

minimizes2 $((\prime a \rightarrow num) \rightarrow (\prime a \rightarrow (\prime a \rightarrow' a)))$ returns `b` if `f(b)` is less than `f(a)`, returns `a` otherwise.

```
 overallcost (times:num->num) (factors:num->num) (n:num) (s:num)
(partition:num array)=
    ( sigma 1 ((asize partition)-2) (* for each batch *)
   (\batchno. batchcost (lookup partition batchno)
              (lookup partition (batchno+1)) times factors n s))
```

overallcost $((num \rightarrow num) \rightarrow ((num \rightarrow num) \rightarrow (num \rightarrow (num \rightarrow ((num)array \rightarrow num)))))$returns the overall cost of `partition` by adding the cost of each batch. Batches are determined by the first job, so the last element of the partition, which is always `n+1`, does not determine a batch.

```
val num_Axiom = theorem "prim_rec" "num_Axiom";
val sigma2_DEF = new_recursive_definition
    {name= "sigma2_DEF", fixity= Prefix, rec_axiom= num_Axiom,
         def= --‘(sigma2 0 i r = (r i)) /\
       (sigma2 (SUC k) i r = (r (i+k)) + (sigma2 k i r))
              ‘--};
    (sigma i j (r:num->num) =  ((i <= j) =>
       (sigma2 (j-i) i r) | 0))
```

sigma$(num \to (num \to ((num \to num) \to num)$ computes $\sum_i^j r$. **sigma2** is the primitive recursive equivalent of it, which is needed since HOL only accepts recursive functions in primitive recursive form.

```
    strictinc (lowb:num) (upb:num) (f:num->num) =
    (!k. (lowb<= k /\ k < (upb - 1)) ==>
              (f k) < (f (k+1)))
```

strictinc$(num \to (num \to ((num \to num) \to bool)))$ returns true if the function `f` is strictly increasing in the range [`lowb`, `upb`).

```
 sub_min_cost_p (times:num->num) (factors:num->num) (j:num) (n:num)
(s:num) (spartition:num array)=
    minimizes (overallcost times factors n s) (vsp j n) spartition
```

sub_min_cost_p$((num \to num) \to ((num \to num) \to (num \to (num \to (num \to ((num)array \to bool))))))$ returns true if `spartition` is a partition containing

returns true if `spartition` has the minimum overallcost amongs all the valid partitions containing the jobs `j` through `n`.

```
 subp (a:('a)array) =
 (ABS_array ((\j:num.(j > 0) => lookup a (j+1)| lookup a 0),((asize a)-1)))
```

subp$(('a)array \to ('a)array)$ returns the array `a` with the element in the first position deleted. The elements in indexes 2 and higher have been shifted one position left. The value stored in position 0 is not changed. The size of the array returned is one less than the size of `a`.

```
    vsp (j:num) (n:num) (spartition:num array) =
    ((asize spartition) > 1) /\
    ((lookup spartition 1) = j) /\
    ((lookup spartition ((asize spartition)- 1)) = (SUC n) ) /\
    (strictinc 1 (asize spartition) (lookup spartition))
```

vsp$(num \rightarrow (num \rightarrow ((num)array \rightarrow bool)$ returns true if `spartition` is a partition containing the jobs `j` through `n`. For this, the size of the array must be greater than 1, the element in the first postition must be `j`, the last element must be `n+1` and the elements of the array should be strictly increasing in the range [1,(`asize spartition`)-1)].

APPENDIX C

THEOREMS

C.1 Auxiliary Theorems

Automatically Proven Theorems

```
nthm : !(n:num). ~((n+1) <= n) /\ ~((n+1) < (n + 1))
LESSEQNOTEQ : !a. (!b. (a <= b) ==>~(a = (b+1)))
GREADD1 : !a. (!b. (a > b) ==>((a+1) >  b))
LESSNOTEQ : !a. (!b. (a > b) ==>~(a = b))
LESSEQLESSEQ : !a. (!b. (a <= b) ==>((a+1) <= (b+1)))
LESSEQLESS : !a. (!b. (a <= b) ==>(a < (b+1)))
GRE10 : 1>0
GREEQ21 : 2>=1
LESS12 : 1<2
GRE21 : !a. (a = 2) ==> (a > 1)
GRELESSGRE10 : !a. (a > 1) ==> ((a-1) > 0)
GRE1GREEQSUB1 : !a. (a > 1) ==> ((a-1) >= 1)
GRELESSGRE21 : !a. (a > 2) ==> ((a-1) > 1)
GRELESSGRE2RW : !a. (a > 2) = ((a-1) >= 2)
GREGRE21 : !a. (a > 2) ==> (a > 1)
GREGREEQ : !a. (!b. (a > b) = (a >= (b+1)))
INT_CONJ : !a. a = a /\ a
GRELESS : !a. (!b. (a > b) = (b < a))
```

```
ADD1GRE : !a. (a < (a+1))
NOTGRE0THEN0 : !a. ~(a > 0) = (a=0)
LESS_EQ_TRANS : !a. (!b. (!c. ((a = b) /\ (c < a))==>(c < b)))
EQORLESSEQ : !(k:num). (!(n:num).
(!(a:num).  ((a<=(n+1)) ==> ( ( (k=(n+1)) \/ ( (a <= k) /\ (k<=n)) ) =
( (a <= k) /\ (k<=(n+1)))))))
GRELESSEQNOTEQ : !(a:num). (!(b:num). (!(c:num).
((a> b) /\ (a <= c)) ==> ( ~(c = b) )))
GRE1GRE1 : !a. !b.
( (a + 1) <= (b + 1) ) = (a <= b)
GRE0SUCSUB : !a. a > 0 ==> (a = SUC(a-1))
GRE1ADD1SUB : !a. a > 1 ==> (a = ((a-1)+1))
GRE1SUB1SUB2 : !a. (a - 1) > 1 ==> (((a-1) -1) = (a-2))
SUB1GRE0SUB1 : !a. !b. 0 < (a-1) ==> ( (a < b)= ((a-1) < (b-1)) )
GRE1SUB1SUB1 : !a. !b. a > 1 ==> ( (a < b)= ((a-1) < (b-1)) )
ALLIMPT : !t. t ==> T
```

Theorems of the Array Theory

```
update_asize:
     !a:('a)array. !i x. (asize (update a i x) = asize a)
update_lookup1:!a:('a)array. !i x.
     (i < asize a) ==> (lookup (update a i x) i = x)
update_lookup2:!a:('a)array. !i x j.
     ~(j = i) ==> (lookup (update a i x) j = lookup a j)
update_id:!a:('a)array. !i. update a i (lookup a i) = a
array_EQ: !a:('a)array. !b. (a = b) =
      (asize a = asize b) /\
      (!i. i < asize a ==> (lookup a i = lookup b i))
```

Theorems Proven In the Scope of The Study

```
merge : !(a:bool). !(b:bool). !(Z:bool).
((a ==> Z ) /\ (b ==> Z)) =
((a \/ b) ==>  Z)
```

```
converter2_THM :
!(n:num) (next:(num) array) (times:num->num)
                        (factors:num->num) (s:num).
((((contains_partition next (n+1) n (init_partition n)) /\
(sub_min_cost_p times factors (n+1) n s (init_partition n)))/\
((overallcost times factors n s (init_partition n))=0)) =
(!k. (k= (n+1)) ==>
(( (contains_partition next k n (init_partition n)) /\
(sub_min_cost_p times factors k n s (init_partition n)))/\
((overallcost times factors n s (init_partition n))=0)
)))

unique1 : !(a:num). !(b:num). !(c:num). !(d:num).
(((( (c > a) /\ (c <= b)) /\ (d > a)) /\ (d <= b)) ==>
(!(f:num->num).
!(k:num). ((k > a) /\ (k <= b)) ==>
     ((((f c) <= (f k))  /\  ((f d) <= (f k)))
   ==>
        ((f c) = (f d)))
))
```

Conjectures involving the addp and subp functions

```
addp_asize : !a:('a)array. !(j:'a). (asize (addp j a)) = ((asize a)+1)

addp_lookup1 : !a:('a)array. !(j:'a).  (lookup (addp j a) 1 = j)

addp_lookup2 : !a:('a)array. !(i:num). !(j:'a).
( (i > 1) ==> ((lookup (addp j a) i) = (lookup a (i-1)) ) )

addp_overallcost :
!a:(num)array. !(times:num->num). !(factors:num->num).
!(n:num).  !(s:num).  !(j:num).
```

```
((overallcost times factors n s (addp j a)) =
((overallcost times factors n s a)
      + (batchcost j (lookup a 1) times factors n s))
)

addp_0 : !a:('a)array. !(j:'a).
(lookup (addp j a) 0) = (lookup a 0)

subp_0 : !a:('a)array. (asize a > 2) ==>
((lookup (subp a) 0) = (lookup a 0))

subp_asize : !a:('a)array.
((asize a > 2) ==> ((asize (subp a)) = ((asize a)-1)))

subp_lookup : !a:('a)array. !(i:num).
((asize a > 2) /\ (i > 0)) ==>
((lookup (subp a) i) = (lookup a (i+1)))

subp_vsp2 : !a:(num)array.
(asize a > 2) ==>
(!(n:num). ((vsp (lookup a 1) n a) ==>
(vsp (lookup (subp a) 1) n (subp a))))

addp_subp : !a:('a)array.
((asize a > 2) ==> (a = addp (lookup a 1) (subp a)))
```

Theorems involving the minimizes, minimizes2, minvalue and minvalue2 predicates

```
!(f:'a->num) (s:'a->bool) (a:'a) (m:'a).
(minimizes f s a ==>
minimizes f (\k. (s k) \/ (k=m)) (minimizes2 f a m))

!(f:'a->num) (s:'a->bool) (a:'a) (m:'a).
```

```
(minimizes f s a ==>
(minvalue f (\k. (s k) \/ (k=m)) (minvalue2 f a m)))

sigma1_THM : !(i:num). !(j:num). !(r:num->num).
(j >= i) ==>
((sigma i j r) = (r i) + (sigma i (j-1) r))

sigma2_THM : !(i:num). !(j:num). !(r:num->num).
(j >= i) ==> ((sigma i j r) = (r i) + (sigma (i+1) j r))

strictinc_THM : !a. !b. !c. !(f:num->num).
(((strictinc a b f) /\ (c >= a)) /\ (c <= (b-1))) ==>
((f c) <= (f (b-1)))

strictinc_btw2_THM : !a. !b. !c. !(f:num->num).
(((strictinc a b f) /\ (a <= c)) /\ (c < (b-1))) ==>
(((f c) < (f (b-1))) /\ ((f a) <= (f c)))

(!(k:num). (!(times:(num) array). (!(factors:(num) array).
(!(s:num). (!(n:num). (!(next:(num) array). (!(costn:(num) array).
(min_costed_next (lookup costn)
    (lookup times) (lookup factors) k n s (lookup next k)) /\
(min_cost (lookup costn)
    (lookup times) (lookup factors) k n s (lookup costn k)) /\
(contains_partition next (lookup next k) n p) /\
(sub_min_cost_p
    (lookup times) (lookup factors) (lookup next k) n s p) /\
((lookup costn (lookup next k)) =
    (overallcost (lookup times) (lookup factors) n s p)) /\
(!x'.  (((SUC k) <= x') /\ (x' <= (n+1))) ==>
    (?p.  (contains_partition next x' n p) /\
       ((sub_min_cost_p
           (lookup times) (lookup factors) x' n s p) /\
```

```
       ((lookup costn x') =
           (overallcost (lookup times) (lookup factors) n s p))))) /\
(k <= n) /\ (k > 0) /\ (x = k) /\
((lookup costn (n + 1)) = 0) /\
((asize next) = (n + 1)) /\
((asize costn) = (n + 2))
==>
(?p'.  (contains_partition next k n p') /\
       (sub_min_cost_p
           (lookup times) (lookup factors) k n s p') /\
       ((lookup costn k) =
           (overallcost (lookup times) (lookup factors) n s p')) /\
       (p' = (addp k p))))))))))
```