Flyspeck II: The Basic Linear Programs

Steven Obua
Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Flyspeck II: The Basic Linear Programs

Steven Obua

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Thomas Huckle

Prüfer der Dissertation: 1. Univ.-Prof. Tobias Nipkow, Ph.D.

2. Prof. Thomas C. Hales
University of Pittsburgh / USA

Die Dissertation wurde am 31. Januar 2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 8. Mai 2008 angenommen.

To Frank Feuerstein,
who was the first to teach me
what mathematics is all about.

Contents

CHAPTER **1**

Introduction

There are two conflicting primal impulses of the human mind – one to simplify a thing to its essentials, the other to see through the essentials to the greater implications.
— Robert B. Laughlin

The *Flyspeck project* [14] has as its goal the complete formalization of Hales' proof [15, 16] of the Kepler conjecture which states that the best density one can hope for when packing infinitely many congruent balls is

$$\frac{\pi}{\sqrt{18}} \approx 0.74$$

The formalization has to be carried out within a mechanical theorem prover. For our work described in this thesis, we have chosen the interactive proof assistant Isabelle [25].

The proof of the Kepler conjecture proceeds in reducing the problem to a finite number of possible counter examples, the *tame graphs*. In a previous research effort [23], the enumeration of all tame graphs has been formalized and verified.

The final computational step of the Kepler conjecture is to prove by linear programming that all of these tame graphs cannot correspond to optimal packings, except those corresponding to the face-centered cubic or hexagonal-close packing.

In this thesis we focus on the *basic linear programs*, which are an important first milestone of taking this final step. With their help, we can eliminate 2565 of the 2771 tame graphs.

How reliable is this result? The major source of potential mayhem is that some mistake might have been introduced in the specification of the basic linear programs. The correctness of this specification will only be established after using the obtained results in the larger context of a complete formal proof of the Kepler conjecture. But even if there is such an error, we can console ourselves that the methods presented in this thesis are general enough so that a transfer to a corrected specification should be possible, and probably easy.

Another potential source of mistakes is the use of the HOL Computing Library which we introduce and describe in the next chapter. After all, it is just a piece of unverified software which has been tested by only one person.

Apart from that, the usual claims of computer-checked proofs hold.

CHAPTER 2

The HOL Computing Library

Fast is fine, but accuracy is everything.
— Wyatt Earp

Contents

2.1 What is the HCL?

The Higher-Order Logic Computing Library (HCL) is an extension of the Isabelle system [25] for fast and trusted computing. Work on it started in 2004 when it became clear that the Flyspeck project [14] demanded a flexible combination of computing power and theorem proving not available in the Isabelle system. Higher-Order Logic (HOL) contains a functional programming language; several research efforts have exposed and exploited this fact. Among these efforts has been to come up with ever more powerful and clever packages for defining functions in this language [33], most recently the function package [19] in Isabelle. The major output of these packages is a list of proven equations about the defined functions which looks very much like a definition of these functions in a functional programming language like Standard ML (SML) [21, 29]. As you are working with a theorem prover, you would like to be able to execute those functions just like in SML, but with the goal of obtaining a theorem or parts of a theorem. Actually, you might want to do so for *any* list of proven equations which look like a functional program, no matter what their origin might be.

The established way of doing this in Isabelle is the Simplifier, which is a tool for doing trusted higher-order rewriting. How does the Simplifier gain your trust? By following a principle that the Edinburgh LCF theorem prover pioneered [9], and which also forms the heart of the Isabelle theorem prover. Theorems are represented as an abstract datatype, and the logic is encoded as operations on this abstract datatype. The Simplifier uses only those operations to generate and manipulate theorems. No matter how complex the implementation of the Simplifier is, all theorems that it generates are guaranteed to be correct as long as the abstract datatype of theorems is implemented correctly.

Obviously there is a price to pay for this combination of power and safety: performance. Compared to running an SML program, the Simplifier incurs a slowdown factor of 1000 and more. In many applications it is worth paying this price, but for the computations in this thesis such a performance penalty is prohibitive. The HCL closes this performance gap, so that computations resulting in theorems are possible with the speed of SML programs. Of course, there is again a price to pay for this performance increase: power and safety. The HCL is less powerful than the Simplifier: it has no congruence rules, but only weaker conditional rewrite rules, and there are no simplification procedures. Currently, the HCL is also not as safe as the Simplifier: there is no formal proof that the implementation of the HCL is correct, therefore you have to trust the HCL just as you trust the implementation of the abstract datatype of theorems. So when you use the HCL, you in fact view it as an extension of the trusted kernel of the Isabelle system. Assuming that the HCL has been implemented correctly, there is no way that using the HCL will produce incorrect theorems [1].

2.2 Arithmetic in Commutative Rings with Unity

Arithmetic is the archetypal computational task and therefore a good first example for computing with the HCL. We choose the abstract setting of commutative rings with unity for doing arithmetic. We introduce an axiomatic type class *Number* which describes such rings. It assumes the presence of the familiar constants and axioms of ring theory (fig. 2.1). All these constants and axioms are polymorphic in α, where α is a type variable of sort *Number*.

We also need a representation for numerals like 67. We adopt the standard approach of Isabelle which is based on [5] and choose a binary representation which can encode both positive and negative numbers. This has the advantage of a uniform addition algorithm which works for any combination of positive and negative operands. Such numerals are built up from four constants (fig. 2.2). Examples are

[1] The true content of this statement probably is: there is a way to implement the HCL correctly.

Name	Type
Zero	α
One	α
add	$\alpha \to \alpha \to \alpha$
mult	$\alpha \to \alpha \to \alpha$
neg	$\alpha \to \alpha$

$$
\begin{array}{lcl}
add\ Zero\ x & = & x \\
add\ x\ y & = & add\ y\ x \\
add\ (add\ x\ y)\ z & = & add\ x\ (add\ y\ z) \\
add\ (neg\ x)\ x & = & Zero \\
mult\ One\ x & = & x \\
mult\ x\ y & = & mult\ y\ x \\
mult\ (mult\ x\ y)\ z & = & mult\ x\ (mult\ y\ z) \\
mult\ x\ (add\ y\ z) & = & add\ (mult\ x\ y)\ (mult\ x\ z)
\end{array}
$$

Figure 2.1: *Number* Constants and Axioms

Name	Type	Definition
Zero	α	
Neg_1	α	$Neg_1 = neg\ One$
B_0	$\alpha \to \alpha$	$B_0\ x = add\ x\ x$
B_1	$\alpha \to \alpha$	$B_1\ x = add\ (add\ x\ x)\ One$

Figure 2.2: Numeral Building Blocks

shown in Figure 2.3.

Negation is a little bit more difficult with this representation compared to a representation where you store the sign separately; it takes linear time instead of constant time. Linear-time addition is also a little bit tricky for the special case when you add two numerals which have both the shape $B_1(\ldots)$. An obvious solution seems to be

$$add\ (B_1\ x)\ (B_1\ y) \quad = \quad B_0\ (add\ (add\ x\ y)\ One),$$

but it is not clear that this is a linear-time rule as computing the successor is worst-case linear-time itself which could lead to a quadratic-time rule for *add*. We define two helper functions neg_1 and add_1 to deal with these difficulties:

$$
\begin{array}{lcl}
neg_1\ x & = & neg\ (add\ x\ One), \\
add_1\ x\ y & = & add\ (add\ x\ y)\ One.
\end{array}
$$

It is then straightforward to prove theorems about negation (fig. 2.4), addition (fig. 2.5) and multiplication (fig. 2.6) which actually look like a functional program for computing these operations on numerals. There are also theorems for normalizing numerals (fig. 2.7).

Decimal	Binary	Numeral
0	0	*Zero*
1	1	$B_1\ Zero$
−1	−1	Neg_1
2	10	$B_0\ (B_1\ Zero)$
−2	−10	$B_0\ Neg_1$
67	1000011	$B_1\ (B_1\ (B_0\ (B_0\ (B_0\ (B_0\ (B_1\ Zero))))))$
−67	−1000011	$B_1\ (B_0\ (B_1\ (B_1\ (B_1\ (B_1\ (B_0\ Neg_1))))))$

Figure 2.3: Numeral Examples

$$
\begin{array}{lcl}
\mathit{neg}\ \mathit{Zero} & = & \mathit{Zero} \\
\mathit{neg}\ \mathit{Neg}_1 & = & B_1\ \mathit{Zero} \\
\mathit{neg}\ (B_0\ x) & = & B_0\ (\mathit{neg}\ x) \\
\mathit{neg}\ (B_1\ x) & = & B_1\ (\mathit{neg}_1\ x)
\end{array}
\qquad
\begin{array}{lcl}
\mathit{neg}_1\ \mathit{Zero} & = & \mathit{Neg}_1 \\
\mathit{neg}_1\ \mathit{Neg}_1 & = & \mathit{Zero} \\
\mathit{neg}_1\ (B_0\ x) & = & B_1\ (\mathit{neg}_1\ x) \\
\mathit{neg}_1\ (B_1\ x) & = & B_0\ (\mathit{neg}_1\ x)
\end{array}
$$

Figure 2.4: Computing Negation

$$
\begin{array}{lcl}
\mathit{add}\ (B_0\ x)\ (B_0\ y) & = & B_0\ (\mathit{add}\ x\ y) \\
\mathit{add}\ (B_0\ x)\ (B_1\ y) & = & B_1\ (\mathit{add}\ x\ y) \\
\mathit{add}\ (B_1\ x)\ (B_0\ y) & = & B_1\ (\mathit{add}\ x\ y) \\
\mathit{add}\ (B_1\ x)\ (B_1\ y) & = & B_0\ (\mathit{add}_1\ x\ y) \\
\mathit{add}\ \mathit{Zero}\ x & = & x \\
\mathit{add}\ x\ \mathit{Zero} & = & x \\
\mathit{add}\ \mathit{Neg}_1\ (B_0\ x) & = & B_1\ (\mathit{add}\ \mathit{Neg}_1\ x) \\
\mathit{add}\ \mathit{Neg}_1\ (B_1\ x) & = & B_0\ x \\
\mathit{add}\ (B_0\ x)\ \mathit{Neg}_1 & = & B_1\ (\mathit{add}\ x\ \mathit{Neg}_1) \\
\mathit{add}\ (B_1\ x)\ \mathit{Neg}_1 & = & B_0\ x \\
\mathit{add}\ \mathit{Neg}_1\ \mathit{Neg}_1 & = & B_0\ \mathit{Neg}_1
\end{array}
\qquad
\begin{array}{lcl}
\mathit{add}_1\ (B_0\ x)\ (B_0\ y) & = & B_1\ (\mathit{add}\ x\ y) \\
\mathit{add}_1\ (B_0\ x)\ (B_1\ y) & = & B_0\ (\mathit{add}_1\ x\ y) \\
\mathit{add}_1\ (B_1\ x)\ (B_0\ y) & = & B_0\ (\mathit{add}_1\ x\ y) \\
\mathit{add}_1\ (B_1\ x)\ (B_1\ y) & = & B_1\ (\mathit{add}_1\ x\ y) \\
\mathit{add}_1\ \mathit{Neg}_1\ x & = & x \\
\mathit{add}_1\ x\ \mathit{Neg}_1 & = & x \\
\mathit{add}_1\ \mathit{Zero}\ (B_0\ x) & = & B_1\ x \\
\mathit{add}_1\ \mathit{Zero}\ (B_1\ x) & = & B_0\ (\mathit{add}_1\ \mathit{Zero}\ x) \\
\mathit{add}_1\ (B_0\ x)\ \mathit{Zero} & = & B_1\ x \\
\mathit{add}_1\ (B_1\ x)\ \mathit{Zero} & = & B_0\ (\mathit{add}_1\ x\ \mathit{Zero}) \\
\mathit{add}_1\ \mathit{Zero}\ \mathit{Zero} & = & B_1\ \mathit{Zero}
\end{array}
$$

Figure 2.5: Computing Addition

$$
\begin{array}{lcl}
\mathit{mult}\ x\ \mathit{Zero} & = & \mathit{Zero} \\
\mathit{mult}\ \mathit{Zero}\ x & = & \mathit{Zero} \\
\mathit{mult}\ \mathit{Neg}_1\ x & = & \mathit{neg}\ x \\
\mathit{mult}\ x\ \mathit{Neg}_1 & = & \mathit{neg}\ x \\
\mathit{mult}\ (B_0\ x)\ y & = & B_0\ (\mathit{mult}\ x\ y) \\
\mathit{mult}\ x\ (B_0\ y) & = & B_0\ (\mathit{mult}\ x\ y) \\
\mathit{mult}\ (B_1\ x)\ (B_1\ y) & = & B_1\ (\mathit{add}\ (B_0\ (\mathit{mult}\ x\ y))\ (\mathit{add}\ x\ y))
\end{array}
$$

Figure 2.6: Computing Multiplication

$$
\begin{array}{lcl}
B_0\ \mathit{Zero} & = & \mathit{Zero} \\
B_1\ \mathit{Neg}_1 & = & \mathit{Neg}_1
\end{array}
$$

Figure 2.7: Normalizing Numerals

```
signature NUMERAL = sig

datatype Numeral = Zero | Neg1
  | B0 of Numeral | B1 of Numeral

val neg : Numeral -> Numeral
val add : Numeral -> Numeral -> Numeral
val mult : Numeral -> Numeral -> Numeral

val norm : Numeral -> Numeral

val test : Numeral -> bool
val fac : Numeral -> Numeral

end

structure Numeral : NUMERAL = struct

datatype Numeral = Zero | Neg1
  | B0 of Numeral | B1 of Numeral

fun add (B0 x) (B0 y) = B0 (add x y)
  | add (B0 x) (B1 y) = B1 (add x y)
  | add (B1 x) (B0 y) = B1 (add x y)
  | add (B1 x) (B1 y) = B0 (add1 x y)
  | add Zero x = x
  | add x Zero = x
  | add Neg1 (B0 x) = B1 (add Neg1 x)
  | add Neg1 (B1 x) = B0 x
  | add (B0 x) Neg1 = B1 (add x Neg1)
  | add (B1 x) Neg1 = B0 x
  | add Neg1 Neg1 = B0 Neg1
and add1 (B0 x) (B0 y) = B1 (add x y)
  | add1 (B0 x) (B1 y) = B0 (add1 x y)
  | add1 (B1 x) (B0 y) = B0 (add1 x y)
  | add1 (B1 x) (B1 y) = B1 (add1 x y)
  | add1 Neg1 x = x
  | add1 x Neg1 = x
  | add1 Zero (B0 x) = B1 x
  | add1 Zero (B1 x) = B0 (add1 Zero x)
  | add1 (B0 x) Zero = B1 x
  | add1 (B1 x) Zero = B0 (add1 x Zero)
  | add1 Zero Zero = B1 Zero

fun neg Zero = Zero
  | neg Neg1 = B1 Zero
  | neg (B0 x) = B0 (neg x)
  | neg (B1 x) = B1 (neg1 x)
and neg1 Zero = Neg1
  | neg1 Neg1 = Zero
  | neg1 (B0 x) = B1 (neg1 x)
  | neg1 (B1 x) = B0 (neg1 x)

fun mult x Zero = Zero
  | mult Zero x = Zero
  | mult Neg1 x = neg x
  | mult x Neg1 = neg x
  | mult (B0 x) y = B0 (mult x y)
  | mult x (B0 y) = B0 (mult x y)
  | mult (B1 x) (B1 y) =
      B1 (add (B0 (mult x y)) (add x y))

fun norm Zero = Zero
  | norm Neg1 = Neg1
  | norm (B0 x) =
     (case norm x of
          Zero => Zero
        | x => B0 x)
  | norm (B1 x) =
     (case norm x of
          Neg1 => Neg1
        | x => B1 x)

fun test Zero = true
  | test Neg1 = false
  | test (B0 x) = test x
  | test (B1 x) = false

fun fac n =
    if test n then
        B1 Zero
    else
        mult n (fac (add n Neg1))

end
```

Figure 2.8: Standard ML Module for Computing with Numerals

2.3 Performance Showdown: Factorials

We will now apply both the HCL and the Simplifier of Isabelle to the problem of computing factorials using the list L of theorems displayed in Figures 2.4, 2.5, 2.6 and 2.7. Both HCL and Simplifier can be used as functions from L to a *conversion*. This conversion is a function taking a term t and returning a theorem $t = F\ t$, where $F\ t$ is the result of rewriting t according to L. To assess the performance of the HCL and the Simplifier we will be looking at a whole family t_i of polymorphic terms where

$$t_0 = N_1 \quad \text{and} \quad t_{i+1} = \mathit{mult}\ t_i\ N_{i+1} \quad .$$

The term N_i is the numeral corresponding to i, e.g. $N_1 = B_1\ (\mathit{Zero} :: \alpha :: \mathit{Number})$, such that it cannot be rewritten further using the theorems in Figure 2.7. In other words, we request that N_i be normalized.

Applying the conversion to t_i will yield the theorem

$$t_i = N_{i!} \quad .$$

We admit one more competitor to this contest: Standard ML itself. The list L looks almost like a functional program, and it is easy to convert it into a true Standard ML program (fig. 2.8). All we need to do is to

1. introduce an SML datatype `Numeral` consisting of four constructors corresponding to the constants *Zero*, Neg_1, B_0 and B_1,
2. introduce SML functions `neg`, `add` and `mult` (together with their helper functions) which correspond to the constants *neg*, *add* and *mult*, and which behave according to Figures 2.4, 2.5, 2.6,
3. introduce an SML function `norm` which does the normalization according to Figure 2.7. There is no Isabelle constant *norm* to which `norm` corresponds, as logically, *norm* would just be the identity.

Additionally, there is a function `test` which checks if the normalization of its input yields `Zero` [2], and there is a function `fac` which calculates the factorial of its input using `test`. Let $\mathtt{N}_i$ be the SML version of N_i, then Standard ML enters the contest by computing `norm (fac N`$_i$`)`.

The results of the performance showdown are summarized in Tables 2.1 and 2.2. The measurements have been taken on an Intel Core2 Duo 2.0 GHz processor running Isabelle 2007 / PolyML 5.0 / GHC 6.6.1. Missing entries in the tables do not indicate nontermination but just that these measurements have not been taken.

Table 2.1 displays the total runtime of each method, while Table 2.2 displays the slowdown factor of how many times slower each method worked relative to just running a functional program. This slowdown factor grows rapidly with the size of the computation for the Simplifier, but it remains constant or even improves for the HCL. The HCL can be operated in different modes. The Barras mode performs interpreted evaluation while both the Haskell mode and the SML mode perform compiled evaluation. The slowdown factor for the Barras mode seems to be between 130 and 90, converging to the better end of this spectrum for large inputs. The Haskell mode uses the external Glasgow Haskell compiler and has therefore huge

[2] `test` could also have been defined via `fun test x = (norm x = Zero)`

i	**Simplifier**	**HCL (Barras)** computing theorem $t_i = N_{i!}$	**HCL (Haskell)**	**HCL (SML)**	**Standard ML** computing SML value `norm (fac N_i)`
10	0.0026	0.0008	0.37	0.00048	0.000006
20	0.016	0.0026	0.38	0.00095	0.000023
40	0.123	0.013	0.4	0.0019	0.00012
80	1.18	0.072	0.45	0.0048	0.0008
160	13.1	0.4	0.54	0.0143	0.0041
320	151	2.3	0.8	0.047	0.024
640	1514	12.4	1.74	0.2	0.13
1280	16400	64.8	5.5	0.91	0.71
2560	–	333	22.5	4.47	3.59
5120	–	1655	102	22.5	18.7
10240	–	–	508	105	102
20480	–	–	–	593	576

Table 2.1: Performance Showdown (runtime in seconds)

i	**Simplifier**	**HCL (Barras)** computing theorem $t_i = N_{i!}$	**HCL (Haskell)**	**HCL (SML)**	**Standard ML** computing SML value `norm (fac N_i)`
10	433	133	61667	80	1
20	696	113	16521	41	1
40	1025	108	3333	16	1
80	1475	90	563	6	1
160	3195	97	38	3.5	1
320	6291	96	17	2	1
640	11646	95	13	1.5	1
1280	23099	91	8	1.3	1
2560	–	93	5	1.2	1
5120	–	89	5.5	1.2	1
10240	–	–	5	1.03	1
20480	–	–	–	1.03	1

Table 2.2: Performance Showdown ($\frac{\text{runtime of method}}{\text{runtime for computing SML value}}$)

test-le Zero	=	*True*	*test-less Zero*	=	*False*
test-le Neg_1	=	*True*	*test-less* Neg_1	=	*True*
test-le (B_0 *x*)	=	*test-le x*	*test-less* (B_0 *x*)	=	*test-less x*
test-le (B_1 *x*)	=	*test-less x*	*test-less* (B_1 *x*)	=	*test-less x*

Figure 2.9: Signs of Integer Numerals

overhead. For long computations though its slowdown factor converges to 5. For the SML mode the numbers are even better: although for small inputs due to non-computational overhead the slowdown factor can be around 80, too, the situation improves dramatically for larger inputs, approaching 1.2 for computations which last a few seconds, and even approaching 1.03 for computations which last minutes.

The road to theorem proving performance that rivals the performance of functional programs is therefore clear: use computing libraries like the HCL, and package computation in as large chunks as possible to avoid the large slowdown factors for small inputs.

2.4 Controlling Evaluation

The observant reader might have noticed that for our performance showdown in the previous section we had an SML function `fac` for computing the factorial of a `Numeral` but no corresponding Isabelle constant *fac*. Instead we worked with an explicit product. This was not really important for the showdown because the computational difference is only slight, but in order to explain some issues concerning the evaluation strategy employed by the HCL, we now introduce such a constant.

However, our current general setting of commutative rings poses difficulties for defining and executing *fac*. A definition like

$$fac\ x = \textbf{if}\ x = Zero\ \textbf{then}\ B_1\ Zero\ \textbf{else}\ mult\ x\ (fac\ (add\ x\ Neg_1))$$

would render *fac* in most rings a partial function, which is inconvenient as HOL is a logic of total functions. While this problem could be solved using the function package [19] for definining partial functions, there is a more serious one. Just based on the axioms from Figure 2.1 we cannot devise an executable test if a ring numeral is equal to *Zero* or not, which is clearly a prerequisite for executing the above specification. For example, the equation B_0 (B_1 *Zero*) = *Zero* holds for any field of characteristic 2, but is false for the ring of integers.

To simplify matters, we therefore leave the general setting of rings and look at the specific setting of the ring of integers. Here it is no problem to define a total function which is constant on negative integers and which behaves like the factorial function on nonnegative integers:

$$fac\ x = \textbf{if}\ test\text{-}le\ x\ \textbf{then}\ B_1\ Zero\ \textbf{else}\ mult\ x\ (fac\ (add\ x\ Neg_1)) \qquad (2.1)$$

The function *test-le* checks if its argument is less than or equal to *Zero*. The theorems for executing it are listed in Figure 2.9. So how do we execute *fac*?

2.4.1 Conditional Rules

Until now all theorems we gave to the HCL had the form of equations. Actually, the HCL accepts theorems that have a more general shape:

$$A_1 \equiv B_1 \Longrightarrow \ldots \Longrightarrow A_m \equiv B_m \Longrightarrow f\ p_1\ p_2 \ldots p_n \equiv T \tag{2.2}$$

The symbols $\equiv$ and $\Longrightarrow$ denote Isabelle's meta notions of equality and implication, respectively. Furthermore, $f\ p_1\ p_2 \ldots p_n$ must be a *linear pattern*, but f may not be a variable.

◂ Definition 2.1 *Linear Pattern*

A pattern *is either*

1. *a variable*
2. *or a term of the form $f\ p_1\ p_2\ \ldots\ p_n$, such that f is a constant and all p_i are patterns.*

A pattern is called linear *if no variable occurs more than once in it.*

Each variable occurring free in T or in any of the A_j or B_j must occur in one of the patterns.

Such a rule instructs the HCL to rewrite a term $f\ q_1\ q_2\ \ldots\ q_n$ by first matching each q_i to the corresponding p_i. If this does not succeed then the rule is ignored. If the matching succeeds then each of the variables in the patterns will be bound. Substituting these variables in T and in each A_j and B_j results in T', A'_j and B'_j. Afterwards the A'_j and B'_j are evaluated by the HCL, resulting in A''_j and B''_j. If for all j the terms A''_j and B''_j are structurally equal, then $f\ q_1\ q_2 \ldots q_n$ is replaced by T', otherwise the rule is ignored.

Therefore, one way of telling the HCL how to execute *fac* is through conditional rules:

$$\begin{array}{rcl} \textit{test-le}\ x & \longrightarrow & \textit{fac}\ x = B_1\ \textit{Zero}, \\ \neg\ \textit{test-le}\ x & \longrightarrow & \textit{fac}\ x = \textit{mult}\ x\ (\textit{fac}\ (\textit{add}\ x\ \textit{Neg}_1)). \end{array}$$

These two rules can be mechanically transformed into

$$\begin{array}{rcl} \textit{test-le}\ x \equiv \textit{True} & \Longrightarrow & \textit{fac}\ x \equiv B_1\ \textit{Zero} \\ (\neg\ \textit{test-le}\ x\ ) \equiv \textit{True} & \Longrightarrow & \textit{fac}\ x \equiv \textit{mult}\ x\ (\textit{fac}\ (\textit{add}\ x\ \textit{Neg}_1)) \end{array}$$

to match the description of rules accepted by the HCL given above.

2.4.2 Strict or Lazy Evaluation?

Instead of splitting the definition of *fac* into two conditional rules it seems more natural to use its definition directly. This raises the question of how to execute the **if-then-else** construct in (2.1). The HCL has no special built-in support for this construct; to it, it is just another function *If* taking three arguments. Therefore we have to provide equations which express the behavior of this *If* constant:

$$\begin{array}{rcl} \textit{If True}\ a\ b & = & a \\ \textit{If False}\ a\ b & = & b \end{array} \tag{2.3}$$

The evaluation of *fac Zero* could then successfully proceed as follows:

$$
\begin{array}{ll}
\underline{\mathit{fac\ Zero}} & \\
\quad \equiv & \textbf{if}\ \underline{\mathit{test\text{-}le\ Zero}}\ \textbf{then}\ B_1\ \mathit{Zero}\ \textbf{else}\ \mathit{mult\ Zero}\ (\mathit{fac}\ (\mathit{add\ Zero\ Neg_1}))\\
\quad \equiv & \underline{\textbf{if}\ \mathit{True}\ \textbf{then}\ B_1\ \mathit{Zero}\ \textbf{else}\ \mathit{mult\ Zero}\ (\mathit{fac}\ (\mathit{add\ Zero\ Neg_1}))}\\
\quad \equiv & B_1\ \mathit{Zero}
\end{array}
$$

Here is a legal, but not terminating evaluation:

$$
\begin{array}{ll}
\underline{\mathit{fac\ Zero}} & \\
\quad \equiv & \textbf{if}\ \mathit{test\text{-}le\ Zero}\ \textbf{then}\ B_1\ \mathit{Zero}\ \textbf{else}\ \mathit{mult\ Zero}\ (\mathit{fac}\ \underline{(\mathit{add\ Zero\ Neg_1})})\\
\quad \equiv & \textbf{if}\ \mathit{test\text{-}le\ Zero}\ \textbf{then}\ B_1\ \mathit{Zero}\ \textbf{else}\ \mathit{mult\ Zero}\ \underline{(\mathit{fac\ Neg_1})}\\
\quad \equiv & \textbf{if}\ \mathit{test\text{-}le\ Zero}\ \textbf{then}\ B_1\ \mathit{Zero}\\
 & \textbf{else}\ \mathit{mult\ Zero}\ (\textbf{if}\ \mathit{test\text{-}le\ Neg_1}\ \textbf{then}\ B_1\ \mathit{Zero}\ \textbf{else}\ \mathit{mult\ Neg_1}\ (\mathit{fac}\ \underline{(\mathit{add\ Neg_1\ Neg_1})}))\\
\quad \equiv & \textbf{if}\ \mathit{test\text{-}le\ Zero}\ \textbf{then}\ B_1\ \mathit{Zero}\\
 & \textbf{else}\ \mathit{mult\ Zero}\ (\textbf{if}\ \mathit{test\text{-}le\ Neg_1}\ \textbf{then}\ B_1\ \mathit{Zero}\ \textbf{else}\ \mathit{mult\ Neg_1}\ \underline{(\mathit{fac}\ (B_0\ \mathit{Neg_1}))})\\
\quad \equiv & \ldots
\end{array}
$$

Which evaluation path will the HCL take? One of the above, or maybe even another, third one?

Actually, this depends on the mode of the HCL. As mentioned earlier, there are different modes of the HCL, most prominently the Barras mode, the SML mode, and the Haskell mode. The Haskell mode performs lazy evaluation and will therefore find a terminating path for evaluating *fac Zero*. Both the Barras and the SML mode evaluate all the arguments that are mentioned on the left hand side of a rule before applying the rule, and will therefore diverge.

The trouble is that depending on the value of the first argument of *If* either the second or the third argument should not be evaluated. There is a way to teach the HCL this special evaluation strategy. If both the Barras and the SML mode need to evaluate all the arguments on the left hand side of a rule before applying the rule, but you do not want the last two arguments of *If* to be evaluated prematurely, then just remove these two arguments from the left hand side and move them to the right hand side!

Applying this to the equations (2.3) yields new equations for *If* which mark the first argument as strict and the last two arguments as lazy:

$$
\begin{array}{lcl}
\mathit{If\ True} & = & \lambda\, a\, b.\ a\\
\mathit{If\ False} & = & \lambda\, a\, b.\ b
\end{array}
\tag{2.4}
$$

The reasons *why* this results in the behavior we wish for depend again on the mode. For the Haskell mode, everything has worked before and will continue to work. For the Barras mode and the SML mode special care has been taken to accommodate the desired behavior. For details on this see the later sections which describe the implementation of each mode.

Note that this method allows us to split the $n = n_{\text{strict}} + n_{\text{lazy}}$ arguments of *any* function into two groups. The first group is made up of the first n_{strict} arguments which are evaluated strictly. The second group is made up of the last n_{lazy} arguments which are evaluated lazily. The Barras mode is more general than the SML mode in that it allows this grouping to vary from rule to rule. In the SML mode it is assumed that there is a fixed grouping for all rules which belong to the same function.

Another, albeit related, application of this feature is to implement *short-circuit* boolean operators, listed in Figure 2.10. The operators given there are defined using the already available corresponding logical operators of Isabelle/HOL. There is no

Name	Definition
And	*And* $x\ y = x \wedge y$
Or	*Or* $x\ y = x \vee y$
Implies	*Implies* $x\ y = x \longrightarrow y$

$$\begin{aligned}
\textit{And True} &= \lambda\, y.\ y \\
\textit{And False} &= \lambda\, y.\ \textit{False} \\
\textit{Or True} &= \lambda\, y.\ \textit{True} \\
\textit{Or False} &= \lambda\, y.\ y \\
\textit{Implies True} &= \lambda\, y.\ y \\
\textit{Implies False} &= \lambda\, y.\ \textit{True}
\end{aligned}$$

Figure 2.10: Short-circuit Boolean Operators of Type $bool \to bool \to bool$

need to define *new* operators to obtain a different evaluation strategy for *existing* operators. But in doing so we are able to use different evaluation strategies *at the same time*.

It is a strength of our approach that via modes pure computing is decoupled from the embedding into a theorem proving environment. The HCL is easily extendable this way. Do you wish that the HCL could evaluate all arguments of a function in parallel? Then just implement a mode that has this feature. Or outsource this task to somebody who is an expert in programming parallel compilers but maybe has no clue about theorem proving.

2.5 Modes of the HCL

So what exactly is a mode of the HCL? It is an implementation of the abstract machine interface we describe in this section.

Expressions in a theorem prover are complicated. There are terms, types embedded in and describing those terms, theorems, sorts of types, assumptions of theorems, meta assumptions of theorems and so on. To deal correctly with these complications is not trivial and mistakes are easily introduced if one handles them directly bypassing the protective layer of the theorem prover kernel. The design of the HCL takes this potential source of mistakes into account and separates the administrative tasks of the computing library from the actual task of computing. The administrative branch of the HCL supports major features of Isabelle like axiomatic type classes, polymorphism, overloading and locales, and will be studied later. In this section we look at our interface for raw computing and at two of its implementations, the Barras mode and the SML mode. There is also a Haskell mode which is similar to the SML mode, and we will mention some of the differences.

2.5.1 An Abstract Machine Interface

Each computing mode can be viewed as a black box. You give it a program and a term, and the black box will return another term which is the result of running the input program on the input term. This black box is what we call our *abstract machine*. To describe it, we need to explain how exactly our terms and programs look like, and what it means to run such programs on such terms.

◂ Definition 2.2 *Abstract Machine Term*

An abstract machine term *is inductively defined via*

$$\text{term} ::= \textit{Var}\ v \mid \textit{Const}\ c \mid \text{term}_1 \cdot \text{term}_2 \mid \lambda\ \text{term} \mid \textit{Computed}\ \text{term}$$

such that $v \in \mathbb{N}$ *and* $c \in \mathbb{Z}$*. A* pure *term is one that does not contain any Computed terms.*

An abstract machine term is just a λ calculus term in de Bruijn index notation [6], with constants and with an additional constructor *Computed*. For specification purposes *Computed* t can be treated just like t; it is a hint for the abstract machine implementation that t has already been computed and needs no further processing. The implementation may ignore this hint.

When we write $t_1 = t_2$ for two abstract machine terms t_1 and t_2 we are referring to the equality naturally arising from the inductive definition of terms.The advantage of using de Bruijn indices is that we do not need to concern ourselves with α equivalence of λ terms.

Definition 2.3 ► *Abstract Machine Pattern*

An abstract machine pattern *is inductively defined via*

$$\text{pattern} ::= \mathit{PVar} \mid \mathit{PConst}\ c\ [\text{pattern}_1, \ldots, \text{pattern}_n]$$

such that $c \in \mathbb{Z}$.

We denote the number of occurrences of *PVar* in a pattern p by $|p|$. An abstract machine pattern is just a compact encoding of certain abstract machine terms. Let us denote the term that corresponds to a given pattern p by $[p]_T$:

$$\begin{array}{lcl} [p]_T & = & [p]_{T,0} \\ [\mathit{PVar}]_{T,i} & = & \mathit{Var}\ i \\ [\mathit{PConst}\ c\ [p_1,\ldots,p_n]]_{T,i} & = & (\ldots(((\mathit{Const}\ c)\cdot q_1)\cdot q_2)\cdots)\cdot q_n \\ & & \text{where } q_j = [p_j]_{T,I(j)} \text{ and } I(j) = i + \sum_{k=j+1}^{n} |p_k| \end{array}$$

So $[p]_T$ arises from p in the obvious way by enumerating the variables from right to left.

The idea is that a pair (p,t) of a pattern p and a term t induces a rewrite rule $[p]_T = t$. We also require that in such a rewrite rule any free variable in t must be bound by p.

Definition 2.4 ► *checkfrees*

$$\begin{array}{lcl} \mathit{checkfrees}\ f\ (\mathit{Var}\ v) & = & v < f \\ \mathit{checkfrees}\ f\ (\mathit{Const}\ c) & = & \mathit{true} \\ \mathit{checkfrees}\ f\ (t_1 \cdot t_2) & = & \mathit{checkfrees}\ f\ t_1 \wedge \mathit{checkfrees}\ f\ t_2 \\ \mathit{checkfrees}\ f\ (\lambda\ t) & = & \mathit{checkfrees}\ (f+1)\ t \\ \mathit{checkfrees}\ f\ (\mathit{Computed}\ t) & = & \mathit{checkfrees}\ f\ t \end{array}$$

The requirement that in a rewrite rule $[p]_T = t$ any free variable in t must be bound by p can be expressed with the formula *checkfrees* $|p|$ t.

Definition 2.5 ► *Abstract Machine Program*

An abstract machine rule *is a triple* $([(a_1,b_1),\ldots,(a_n,b_n)],p,t)$ *such that*

1. p *is an abstract machine pattern and* $p \neq \mathit{PVar}$,
2. t *is a pure abstract machine term,*
3. *checkfrees* $|p|$ t *holds,*
4. *the* a_i *and* b_i *are pure abstract machine terms with checkfrees* $|p|$ a_i *and checkfrees* $|p|$ b_i*, respectively. The pairs* (a_i,b_i) *are called* guards.

An abstract machine program *is a list of abstract machine rules.*

Now that we know what an abstract machine program looks like, how does it operate on a term? Although we assume familiarity with de Bruijn indices [6] and the λ calculus, we will first define β reduction and related notions for abstract machine terms so that we have the complete definition of the abstract machine interface in one place.

Definition 2.6 *Lifting*

$$
\begin{aligned}
(\mathit{Var}\ v)\uparrow^n &= \begin{cases} \mathit{Var}\ v & \text{if } v < n \\ \mathit{Var}\ (v+1) & \text{if } v \geq n \end{cases}\\
(\mathit{Const}\ c)\uparrow^n &= \mathit{Const}\ c\\
(t_1 \cdot t_2)\uparrow^n &= (t_1\uparrow^n)\cdot(t_2\uparrow^n)\\
(\lambda\ t)\uparrow^n &= \lambda(t\uparrow^{n+1})\\
(\mathit{Computed}\ t)\uparrow^n &= \mathit{Computed}\ (t\uparrow^n)
\end{aligned}
$$

Definition 2.7 *Lowering*

$$
\begin{aligned}
(\mathit{Var}\ v)\downarrow_n &= \begin{cases} \mathit{Var}\ v & \text{if } v < n \\ \mathit{Var}\ (v-1) & \text{if } v > n \end{cases}\\
(\mathit{Const}\ c)\downarrow_n &= \mathit{Const}\ c\\
(t_1 \cdot t_2)\downarrow_n &= (t_1\downarrow_n)\cdot(t_2\downarrow_n)\\
(\lambda\ t)\downarrow_n &= \lambda(t\downarrow_{n+1})\\
(\mathit{Computed}\ t)\downarrow_n &= \mathit{Computed}\ (t\downarrow_n)
\end{aligned}
$$

Let now ζ be a *substitution*, i.e. a partial function from $\mathbb{N}$ to abstract machine terms. The substitution $\zeta\uparrow$ is then defined by

$$
(\zeta\uparrow)\ v = \begin{cases} \text{undefined} & \text{if } v = 0 \text{ or } v > 0\ \wedge\ \zeta(v-1) \text{ is undefined} \\ \zeta(v-1)\uparrow^0 & \text{if } v > 0\ \wedge\ \zeta(v-1) \text{ is defined} \end{cases}
$$

Definition 2.8 *Substitution*

$$
\begin{aligned}
(\mathit{Var}\ v)[\zeta] &= \begin{cases} \zeta\ v & \text{if } \zeta\ v \text{ is defined} \\ \mathit{Var}\ v & \text{if } \zeta\ v \text{ is undefined} \end{cases}\\
(\mathit{Const}\ c)[\zeta] &= \mathit{Const}\ \mathrm{c}\\
(t_1 \cdot t_2)[\zeta] &= (t_1[\zeta])\cdot(t_2[\zeta])\\
(\lambda\ t)[\zeta] &= \lambda\ (t[\zeta\uparrow])\\
(\mathit{Computed}\ t)[\zeta] &= \mathit{Computed}\ (t[\zeta])
\end{aligned}
$$

If ζ is a function just defined for a single index v such that $\zeta\ v = s$ we also write $t[s/v]$ instead of $t[\zeta]$.

Definition 2.9 *β Reduction*

1. $(\lambda\ t)\ s \rightarrow_\beta (t[(s\uparrow^0)/0])\downarrow_0$
2. $\lambda\ t \rightarrow_\beta \lambda\ t'$ if $t \rightarrow_\beta t'$
3. $t_1 \cdot t_2 \rightarrow_\beta t_1' \cdot t_2$ if $t_1 \rightarrow_\beta t_1'$
4. $t_1 \cdot t_2 \rightarrow_\beta t_1 \cdot t_2'$ if $t_2 \rightarrow_\beta t_2'$
5. $\mathit{Computed}\ t \rightarrow_\beta \mathit{Computed}\ t'$ if $t \rightarrow_\beta t'$

Lowering is actually a partial function because e.g. $(\mathit{Var}\ 0)\downarrow_0$ is not defined; but we use it in the definition of β reduction only on an argument for which it is defined.

By defining β reduction we put one aspect of computing in place. On top of it we can build the other aspect, which is how to make use of the rules of an abstract machine program. For this we define τ reduction. When talking about τ reduction, we always have some specific abstract machine program in mind which does not appear explicitly in our notation but should be clear from the context.

Definition 2.10 ► *τ Reduction*

1. $t_1 \Rightarrow_\tau t_2$ if $t_1 \rightarrow_\tau^* t_2$
2. $t_1 =_\tau t_2$ if $\exists s.\ t_1 \Rightarrow_\tau s \wedge t_2 \Rightarrow_\tau s$
3. $([p]_T)[\zeta] \rightarrow_\tau t[\zeta]$ if
   1. ζ is a substitution
   2. and $([(a_1,b_1),\ldots,(a_n,b_n)],p,t)$ is a rule of the program
   3. and $a_i[\zeta] =_\tau b_i[\zeta]$ for all $i = 1,\ldots,n$
4. $t \rightarrow_\tau t'$ if $t \rightarrow_\beta t'$ or $t' \rightarrow_\beta t$ (β conversion)
5. $t \rightarrow_\tau \lambda\ (t \uparrow^0 \cdot \mathit{Var}\ 0)$ and $\lambda\ (t \uparrow^0 \cdot \mathit{Var}\ 0) \rightarrow_\tau t$ (η conversion)
6. $\lambda\ t \rightarrow_\tau \lambda\ t'$ if $t \rightarrow_\tau t'$
7. $t_1 \cdot t_2 \rightarrow_\tau t_1' \cdot t_2$ if $t_1 \rightarrow_\tau t_1'$
8. $t_1 \cdot t_2 \rightarrow_\tau t_1 \cdot t_2'$ if $t_2 \rightarrow_\tau t_2'$
9. $\mathit{Computed}\ t \rightarrow_\tau \mathit{Computed}\ t'$ if $t \rightarrow_\tau t'$
10. $\mathit{Computed}\ t \rightarrow_\tau t$

Until now we have treated the *Computed* constructor just as the identity. We call an abstract machine term t *closed* if *checkfrees* $0\ t$ holds and for any subterm *Computed* s of t the following assumptions hold:

- s is a pure term, i.e. it does not contain *Computed*,
- *checkfrees* $0\ s$ holds,
- there is no s' with $s \rightarrow_\beta s'$.

Definition 2.11 ► *Abstract Machine*

An abstract machine *is a mapping that takes an abstract machine program to a relation $\rightarrow_{AM}$ on closed abstract machine terms such that $\rightarrow_{AM}$ is a partial function and such that $t \rightarrow_{AM} t'$ implies*

- $t \Rightarrow_\tau t'$,
- *t' is a pure term,*
- *there is no t'' such that $t' \rightarrow_\beta t''$.*

Note that an abstract machine is allowed to always fail , i.e. to map every program to the empty relation. This particular abstract machine would not be very useful, but our focus is on ensuring that if an abstract machine *does* compute something, it will be correct. The definition of what constitutes an abstract machine is designed to fulfill this promise while at the same time preserving some freedom for the abstract machine implementor. So when computing a term t an abstract machine can assume that t contains no free variables, and that subterms of t marked with *Computed* have no β redexes left. It must ensure when returning a result t' that any computing done can be understood in terms of τ reduction, that t' contains no *Computed* markers any more, and that there are no β redexes left in t'.

2.5.2 The Barras Machine

The most general implementation of the abstract machine interface that the HCL provides is the *Barras machine*. It is an interpreter with an execution model that is borrowed from the machine in [2]. The most important difference between ours and the original one is that the original one actually produces proofs by operating on theorems instead of terms. Because of this no proof of correctness is given in [2]; we will provide a proof of partial correctness which shows that the Barras machine is a correct implementation of the abstract machine interface. Another difference is that we also allow guards.

The Barras machine mimicks the evaluation strategy of strict functional programming languages. It performs bottom-up evaluation; β reductions are delayed via explicit substitutions.

The data structure of Barras terms is the one for abstract machine terms augmented with an additional constructor *Closure* for performing explicit substitutions. The *Computed* constructor is inherited from the definition of abstract machine terms and does not appear in [2].

$$\begin{aligned}\text{term} ::= {} & \mathit{Var}\ v \mid \mathit{Const}\ c \mid \text{term}_1 \cdot \text{term}_2 \mid \lambda\ \text{term} \mid \mathit{Computed}\ \text{term} \\ & \mid \mathit{Closure}\ [\text{term}_0, \ldots, \text{term}_n]\ \text{term}\end{aligned}$$

◄ Definition 2.12 *Barras Term*

Any abstract machine term is also a Barras term. On the other hand, any Barras term can be understood as an abstract machine term by viewing *Closure* as a function defined by

$$\mathit{Closure}\ [s_1, \ldots, s_n]\ t \quad := \quad (\underbrace{\lambda \ldots \lambda}_{n \text{ times}}\ t) \cdot s_n \cdot \ldots \cdot s_1$$

instead of viewing *Closure* as a constructor.

The idea of the *Closure* constructor is to delay actual β reduction until all arguments of a function have been collected. An additional intuition is that in a term of the form *Closure* E t the *Closure* constructor acts as a marker that t has not been computed yet.

The state (s, t) of the Barras machine consists of the subterm t that is currently reduced and a *stack* s that keeps track of the position of this subterm in the larger term under consideration.

$$\text{stack} ::= \mathit{SEmpty} \mid \mathit{SAppL}\ \text{term stack} \mid \mathit{SAppR}\ \text{term stack} \mid \mathit{SAbs}\ c\ \text{stack}$$

◄ Definition 2.13 *Barras Stack*

Let us denote the larger term that (s, t) encodes with $(s, t)_{\text{zoom out}}$:

$$\begin{aligned}(\mathit{SEmpty},\ t)_{\text{zoom out}} &= \text{t} \\ (\mathit{SAppL}\ t_2\ s,\ t_1)_{\text{zoom out}} &= (s,\ t_1 \cdot t_2)_{\text{zoom out}} \\ (\mathit{SAppR}\ t_1\ s,\ t_2)_{\text{zoom out}} &= (s,\ t_1 \cdot t_2)_{\text{zoom out}} \\ (\mathit{SAbs}\ c\ s,\ t)_{\text{zoom out}} &= (s,\ \lambda\ (t[(\mathit{Var}\ 0)/(\mathit{Const}\ c)]))_{\text{zoom out}}\end{aligned}$$

In the above we substitute a variable for a constant. Let us give a definition for this:

$$\begin{aligned}(\mathit{Const}\ d)[(\mathit{Var}\ v)/(\mathit{Const}\ c)] &= \begin{cases}\mathit{Var}\ v & \text{if } c = d \\ \mathit{Const}\ d & \text{if } c \neq d\end{cases} \\ (\mathit{Var}\ w)[(\mathit{Var}\ v)/(\mathit{Const}\ c)] &= \mathit{Var}\ w \\ (t_1 \cdot t_2)[(\mathit{Var}\ v)/(\mathit{Const}\ c)] &= (t_1\ [(\mathit{Var}\ v)/(\mathit{Const}\ c)]) \cdot (t_2\ [(\mathit{Var}\ v)/(\mathit{Const}\ c)]) \\ (\lambda\ t)[(\mathit{Var}\ v)/(\mathit{Const}\ c)] &= \lambda\ (t\ [(\mathit{Var}\ (v+1))/(\mathit{Const}\ c)]) \\ (\mathit{Closure}\ [e_1, \ldots, e_n]\ t)\ [(\mathit{Var}\ v)/(\mathit{Const}\ c)] &= \mathit{Closure}\ [e'_1, \ldots, e'_n]\ (t\ [(\mathit{Var}\ (v+n))/(\mathit{Const}\ c)]) \\ &\ \text{where } e'_i = e_i\ [(\mathit{Var}\ v)/(\mathit{Const}\ c)] \\ (\mathit{Computed}\ t)[(\mathit{Var}\ v)/(\mathit{Const}\ c)] &= \mathit{Computed}\ (t\ [(\mathit{Var}\ v)/(\mathit{Const}\ c)])\end{aligned}$$

◄ Definition 2.14 *Substituting a Variable for a Constant*

The Barras machine performs both strong and weak reduction. Weak reduction will not reduce those terms $\lambda\ t$ which are not applied to an argument; with strong reduction, if t reduces to t', then $\lambda\ t$ will reduce to $\lambda\ t'$. Because functional programming languages only perform weak reduction, the Barras machine will only perform strong reduction when no further weak reduction is possible.

Definition 2.15 ► *Weak Reduction*

1. $weak\ (s,\ Closure\ E\ (t_1 \cdot t_2)) = weak\ (SAppL\ (Closure\ E\ t_2)\ s,\ Closure\ E\ t_1)$
2. $weak\ (SAppL\ t'\ s,\ Closure\ [e_1,\ldots,e_n]\ (\lambda\ t)) = weak\ (s,\ Closure\ [t',e_1,\ldots,e_n]\ t)$
3. $weak\ (s,\ Closure\ [e_0,\ldots,e_n]\ (Var\ v)) = weak\ (s,\ e_v)$
4. $weak\ (s,\ Closure\ E\ (Const\ c)) = weak\ (s,\ Const\ c)$
5. $weak\ (s,\ Closure\ E\ (Computed\ t)) = \begin{cases} weak\ (s,\ Closure\ []\ t) & \text{if } t \text{ contains any } \lambda \\ weak\ (s,\ t) & \text{otherwise} \end{cases}$
6. $weak\ (s,\ t) = \begin{cases} weak\ (s,r) & \text{if } match\ t = Some\ r \\ weak'\ (s,t) & \text{if } match\ t = None \end{cases}$
7. $weak'\ (SAppR\ t_1\ s,\ t_2) = weak\ (s,\ t_1 \cdot t_2)$
8. $weak'\ (SAppL\ t_2\ s,\ t_1) = weak\ (SAppR\ t_1\ s,\ t_2)$
9. $weak'\ (s,\ t) = (s,\ t)$

The reduction rules should be read like an ML function definition, i.e. rule i will only be applied if there is no applicable rule j with $j < i$. Because of guards, strong and weak reduction are mutually recursive; above rules need a definition of the *match* operation to be complete, and *match* depends on strong reduction. Therefore we first describe strong reduction, and look only then at the *match* operation.

Definition 2.16 ► *Strong Reduction*

1. $strong\ (s,\ Closure\ [e_1,\ldots,e_n]\ (\lambda\ t)) = strong\ (SAbs\ c\ s,\ t')$
   where a) $c \in \mathbb{Z}$ is some fresh identifier not referring to any constant in $e_1, \ldots, e_n$ or in t or in the abstract machine program
   b) $weak\ (SEmpty,\ Closure\ [Const\ c, e_1,\ldots, e_n]\ t) = (SEmpty,\ t')$
2. $strong\ (s,\ t_1 \cdot t_2) = strong\ (SAppL\ t_2\ s,\ t_1)$
3. $strong\ (s,\ t) = strong'\ (s,\ t)$
4. $strong'\ (SAbs\ c\ s,\ t) = strong'\ (s,\ \lambda\ (t[(Var\ 0)/(Const\ c)]))$
5. $strong'\ (SAppL\ t_2\ s,\ t_1) = strong\ (SAppR\ t_1\ s,\ t_2)$
6. $strong'\ (SAppR\ t_1\ s,\ t_2) = strong'\ (s,\ t_1 \cdot t_2)$
7. $strong'\ (s,\ t) = (s,\ t)$

Definition 2.17 ► *Simplification*

Let simp be the partial function defined by

$$simp\ t = t' \quad \text{if} \quad weak\ (SEmpty, t) = (SEmpty, t'') \text{ and } strong\ (SEmpty, t'') = (SEmpty, t').$$

Definition 2.18 ► *Matching*

Let the abstract machine rule

$$([(a_1,b_1),\ldots,(a_n,b_n)],\ p,\ t)$$

be the first rule of the abstract machine program such that

1. $[p]_T[\zeta] = m$ *for some substitution* $\zeta = (0 \mapsto e_0,\ldots,|p|-1 \mapsto e_{|p|-1})$,
2. $E = [e_0,\ldots,e_{|p|-1}]$,
3. $simp\ (Closure\ E\ a_i) = simp\ (Closure\ E\ b_i)$ *for all* $i = 1,\ldots,n$.

Then we define

$$match\ m = Some\ (Closure\ E\ t),$$

otherwise $match\ m = None$.

◂ Definition 2.19 *Barras Machine*

The Barras machine *maps an abstract machine program to the relation* $\to_{\text{Barras}}$ *where*

$$t \to_{\text{Barras}} t' \quad \textit{if} \quad \textit{simp}\ (\textit{Closure}\ []\ t) = t'.$$

◂ Theorem 2.1 *Partial Correctness of the Barras Machine*

The Barras machine is an abstract machine in the sense of Definition 2.11.

Proof. The $\to_{\text{Barras}}$ relation is defined in terms of *simp*, therefore it is a partial function. Let us assume that $t_0 \to_{\text{Barras}} t_F$ holds. Then the machine state $(\textit{SEmpty},\ t_0)$ has been transformed into the state $(\textit{SEmpty},\ t_F)$ via a series of calls to *weak*, *weak'*, *strong* and *strong'*. Showing $t_0 \Rightarrow_\tau t_F$ is equivalent to showing $(\textit{SEmpty},\ t_0)_{\text{zoom out}} \Rightarrow_\tau (\textit{SEmpty},\ t_F)_{\text{zoom out}}$. Note that when talking about τ reduction of Barras terms we view them as abstract machine terms by erasing all closures as explained earlier. Because $\Rightarrow_\tau$ is transitive, $t_0 \Rightarrow_\tau t_F$ follows if we can show $(s_1,\ t_1)_{\text{zoom out}} \Rightarrow_\tau (s_2,\ t_2)_{\text{zoom out}}$ for all consecutive states of the transformation of $(\textit{SEmpty},\ t_0)$ into $(\textit{SEmpty},\ t_F)$. We can assume for any state (s,t) participating in the transformation the following invariant:

- t is closed (when viewed as an abstract machine term),
- any term appearing in the stack s is closed,
- for any subterm c of t or any term appearing in the stack such that c is of shape $\textit{Closure}\ [e_1,\ldots,e_n]\ t'$, we have that c and all e_i are closed,
- if $\textit{SAbs}\ c\ s'$ appears somewhere in s then $\textit{Const}\ c$ does not appear anywhere in the abstract machine program.

This invariant is true for our initial state $(\textit{SEmpty},\ t_0)$ as there are no terms in *SEmpty* and t_0 is a closed abstract machine term; both weak and strong reduction preserve this invariant. This is proven simultaneously with the compatibility of weak and strong reduction with τ reduction, but we do not mention this explicitly.

Why do we need that last condition in our invariant? Because it ensures compatibility of τ reduction with zooming out. All the time we will want to deduce from $t \Rightarrow_\tau t'$ that also $(s,t)_{\text{zoom out}} \Rightarrow_\tau (s,t')_{\text{zoom out}}$ holds. If $s = \textit{SAbs}\ c\ s'$, then this is equivalent to $(s', t[(\textit{Var}\ 0)/(\textit{Const}\ c)])_{\text{zoom out}} \Rightarrow_\tau (s', t'[(\textit{Var}\ 0)/(\textit{Const}\ c)])_{\text{zoom out}}$. But does $t[(\textit{Var}\ 0)/(\textit{Const}\ c)] \Rightarrow_\tau t'[(\textit{Var}\ 0)/(\textit{Const}\ c)]$ hold? Yes, because $\textit{Const}\ c$ does not appear in the abstract machine program. That way, $t \Rightarrow_\tau t'$ is not due to any special properties that $\textit{Const}\ c$ has with respect to the program. By induction it is therefore easy to show that τ reduction is compatible with zooming out.

Let us now check that τ reduction is compatible with each weak reduction rule.

1. From the definition of zooming out we conclude $(\textit{SAppL}\ (\textit{Closure}\ E\ t_2)\ s,\ \textit{Closure}\ E\ t_1)_{\text{zoom out}} = (s,\ (\textit{Closure}\ E\ t_1) \cdot (\textit{Closure}\ E\ t_2))_{\text{zoom out}}$. Thus the compatibility follows from $(\textit{Closure}\ E\ t_1) \cdot (\textit{Closure}\ E\ t_2) \Rightarrow_\tau \textit{Closure}\ E\ (t_1 \cdot t_2)$.
2. $(\textit{SAppL}\ t'\ s,\ \textit{Closure}\ [e_1,\ldots,e_n]\ (\lambda\ t))_{\text{zoom out}} = (s,\ (\textit{Closure}\ [e_1,\ldots,e_n]\ (\lambda\ t)) \cdot t')_{\text{zoom out}}$, therefore the compatibility of rule 2 is a consequence of

$$\begin{aligned}(\textit{Closure}\ [e_1,\ldots,e_n]\ (\lambda\ t)) \cdot t' &= ((\underbrace{\lambda \ldots \lambda}_{n \text{ times}}\ (\lambda\ t)) \cdot e_n \cdot \ldots \cdot e_1) \cdot t' \\ &= (\underbrace{\lambda \ldots \lambda}_{(n+1) \text{ times}}\ t) \cdot e_n \cdot \ldots \cdot e_1 \cdot t' \\ &= \textit{Closure}\ [t', e_1,\ldots,e_n]\ t.\end{aligned}$$

3. The invariant tells us that *Closure* $[e_0,\ldots,e_n]$ (*Var* v) is closed. Thus $v \le n$ and
$$\textit{Closure}\,[e_0,\ldots,e_n]\,(\textit{Var}\,v) \quad = \quad (\lambda\ldots\lambda\,(\textit{Var}\,v))\cdot e_n\cdot\ldots\cdot e_0 \quad \Rightarrow_\tau \quad e_v.$$
4. Similarly, *Closure* E (*Const* c) $\Rightarrow_\tau$ *Const* c.
5. The invariant ensures that in *Closure* E (*Computed* t) the term t is closed. Therefore
$$\textit{Closure}\,E\,(\textit{Computed}\,t) \quad \Rightarrow_\tau \quad \textit{Closure}\,E\,t \quad \Rightarrow_\tau \quad t.$$
6. If no rule matches then the transformation is just the identity and there is nothing to show. On the other hand, assume that the state (s,t) is transformed into (s,r) by matching with the rule $([(a_1,b_1),\ldots,(a_n,b_n)],\ p,\ u)$. Then
   (a) $[p]_T[\zeta] = t$ for some substitution $\zeta = (0 \mapsto e_0,\ldots,|p|-1 \mapsto e_{|p|-1})$,
   (b) $E = [e_0,\ldots,e_{|p|-1}]$,
   (c) *simp* (*Closure* $E\ a_i$) = *simp* (*Closure* $E\ b_i$) for all $i = 1,\ldots,n$,

   and $r =$ *Closure* $E\ u$.

   Inductively, we derive *Closure* $E\ a_i =_\tau$ *Closure* $E\ b_i$ for all $i = 1,\ldots,n$. Because u and all a_i and b_i contain no free variables except those for which ζ is defined, we have $u[\zeta] \Rightarrow_\tau$ *Closure* $E\ u$ and $a_i[\zeta] \Rightarrow_\tau$ *Closure* $E\ a_i$ and $b_i[\zeta] \Rightarrow_\tau$ *Closure* $E\ b_i$. Thus rule 2 of Definition 2.10 for τ reduction is applicable and gives us $t = [p]_T[\zeta] \rightarrow_\tau u[\zeta] \Rightarrow_\tau r$.

The compatibility of rules 7 and 8 follows directly from the definition of zooming out, for rule 9 there is nothing to show.

Next are the strong reduction rules.

1. $(\textit{SAbs}\ c\ s,\ t')_{\text{zoom out}} = (s,\ \lambda\ t'[(\textit{Var}\ 0)/(\textit{Const}\ c)])_{\text{zoom out}}$. The term t' is the result of weakly reducing *Closure* $[\textit{Const}\ c, e_1,\ldots,e_n]\ t$ and thus by induction
$$\textit{Closure}\,[\textit{Const}\,c,e_1,\ldots,e_n]\,t \quad \Rightarrow_\tau \quad t'.$$
   From there we deduce
$$(\textit{Closure}\,[\textit{Const}\,c,e_1,\ldots,e_n]\,t)\,[(\textit{Var}\,0)/(\textit{Const}\,c)] \quad \Rightarrow_\tau \quad t'[(\textit{Var}\,0)/(\textit{Const}\,c)]$$
   because *Const* c does not appear anywhere in the abstract machine program. Furthermore, *Const* c does not appear in $e_1,\ldots,e_n$ and neither in t. Thus
$$\begin{array}{rcl}
\lambda\,t'[(\textit{Var}\,0)/(\textit{Const}\,c)] & \Leftarrow_\tau & \lambda\,((\textit{Closure}\,[\textit{Const}\,c,e_1,\ldots,e_n]\,t)[(\textit{Var}\,0)/(\textit{Const}\,c)]) \\
& = & \lambda\,(\textit{Closure}\,[\textit{Var}\,0,e_1,\ldots,e_n]\,t) \\
& = & \lambda\,((\underbrace{\lambda\ldots\lambda}_{\text{(n+1) times}}\,t)\cdot e_n\cdot\ldots\cdot e_1\cdot \textit{Var}\,0) \\
& = & \lambda\,((\textit{Closure}\,[e_1,\ldots,e_n]\,(\lambda\,t))\cdot \textit{Var}\,0) \\
& = & \lambda\,((\textit{Closure}\,[e_1,\ldots,e_n]\,(\lambda\,t))\uparrow^0\cdot \textit{Var}\,0) \\
& \Leftarrow_\tau & \textit{Closure}\,[e_1,\ldots,e_n]\,(\lambda\,t).
\end{array}$$
4. Follows directly from the definition of zooming out.

Zooming out leads to immediate proofs for all other strong reduction rules. □

Actually, we are not done yet with the proof. To complete the proof that the Barras machine is a true abstract machine, we also have to show that from $t_0 \rightarrow_{\text{Barras}} t_F$ it follows that t_F contains no β-redexes, no *Computed* terms, and no closures anymore.

Proof. For the purpose of this part of the proof, we do not freely interchange Barras terms and abstract machine terms any longer; this means that we view *Closure* as a constructor now, and not as an abbreviation for a special kind of abstract machine term.

We say that a Barras term w is in *weak normal form (WNF)* if it meets all of the following conditions:

1. If w contains λs then those are always contained in a surrounding closure.
2. w contains no closures except those of shape *Closure* E $(\lambda\ t)$.
3. w does not contain any term of shape (*Closure* E $(\lambda\ t)) \cdot s$.
4. If w contains β-redexes then those are always contained in a surrounding closure.
5. If w contains *Computed* terms then those are always contained in a surrounding closure.

We first look at weak reduction and prove by induction over the number of reduction steps that its execution preserves the following invariants of the state (S,T) of the Barras machine:

(I1) If *Closure* $[e_1,\ldots,e_n]$ t is contained in T or in any of the terms in the stack S, then each e_i is a closure or a WNF, and t contains no closures.

(I2) T and all terms in S are closures or WNFs.

(I3) If *SAppL* t s is contained in S then t is a closure.

(I4) If *SAppR* t s is contained in S then t is a WNF.

It is easy to see that all the rules of reduction for *weak* and *weak′* preserve these four invariants:

- Let us start with (I3). *SAppL* t s is only produced in rule 1, and t is clearly a closure there.
- To show (I4), note that *SAppR* t s is only produced in rule 8; t cannot be a closure because otherwise one of the rules 1 to 5 would have been called; because of (I1) these rules form a complete case distinction on closures (at least for an *SAppL* stack). Because of (I2) it follows that t is a WNF.
- It is time to approach (I1). The only critical rules are 2 and 6. Rule 2 preserves (I1) because of (I3). Rule 6 uses the fact that if *match* t succeeds then t cannot be a closure and must therefore be a WNF because of (I2).
- Critical for (I2) are the rules 5 and 7. Rule 5 is defined via a case distinction just so that (I2) is evident. In rule 7 the term $t_1 \cdot t_2$ is a WNF because t_1 is a WNF but no closure (because of (I4)) and because t_2 is also a WNF (t_2 cannot be a closure and no WNF as otherwise one of the rules 1,3,4,5 had applied).

Provided the start state of weak reduction fulfills all four invariants it follows that the terminal state (*SEmpty*, t) fulfills the invariants. Therefore t is a closure or a WNF. It cannot be a closure and not a WNF because then rule 9 would never have applied. Therefore t is a WNF.

Now we look at strong reduction. We introduce two further normal forms. The *weaker normal form (WRNF)* is a still weaker form of the WNF which we obtain by not requiring property 1. A Barras term has *strong normal form (SNF)* if it does not contain any closures.

The invariants for the state (S,T) during strong reduction are then:

(J1) If *Closure* $[e_1, \ldots, e_n]$ t is contained in T or in any of the terms in the stack S, then each e_i is a closure or a WNF, and t contains no closures.

(J2) $(S, T)_{\text{zoom out}}$ is a WRNF.

(J3) In every call *strong* (S, T), T is a WNF.

(J4) In every call *strong′* (S, T), T is a SNF.

(J5) If *SAppL* $t\ s$ is contained in S then t is a WNF.

(J6) If *SAppR* $t\ s$ is contained in S then t is a SNF.

We want to prove that these invariants hold throughout strong reduction by induction over the number of *strong* and *strong′* reduction steps. As induction base, consider how strong reduction is called indirectly via *simp* (*Closure* [] t): first *weak* (*SEmpty*, *Closure* [] t) = (*SEmpty*, t'') leads to t'' and then *strong* (*SEmpty*, t'') is called. The state (*SEmpty*, *Closure* [] t) fulfills trivially all invariants I1 to I4 and therefore t'' is a WNF as we have proved before. This implies that all invariants J1 to J6 hold trivially for the call *strong* (*SEmpty*, t'').

The induction step is again easy to do; we just check that all the invariants stay true:

- Let us consider rule 1 of strong reduction. A call to

  $$\textit{strong}\ (s,\ \textit{Closure}\ [e_1, \ldots, e_n]\ (\lambda\ t))$$

  results in a call to *strong* (*SAbs* $c\ s$, t') where t' is the result of

  $$\textit{weak}\ (\textit{SEmpty},\ \textit{Closure}\ [\textit{Const}\ c, e_1, \ldots, e_n]\ t)\ =\ (\textit{SEmpty},\ t').$$

  All invariants I1 to I4 are true for the state *weak* is called with: I1 holds because J1 holds for (s, *Closure* $[e_1, \ldots, e_n]$ $(\lambda\ t)$) and because *Const* c is a WNF, similarly we derive I2 from J3. I3 and I4 hold trivially because the stack is empty.

  Thus the result of weak reduction, t', is a WNF. Therefore J3 holds. There is no call to *strong′* which implies that J4 holds trivially. All closures appearing in t' have to respect I1 and therefore also J1 holds. The stack *SAbs* $c\ s$ inherits all *SAppL* and *SAppR* elements from s, therefore J5 and J6 hold trivially. The only invariant we have to be a little bit careful about is J2. In order for $(\textit{SAbs}\ c\ s,\ t')_{\text{zoom out}}$ to be a WRNF, we must ensure that it does not contain any β-redexes. Zooming one step out of (*SAbs* $c\ s$, t') yields a state of the form (s, $\lambda \ldots$), so there is the danger of introducing such a β-redex if s is an *SAppL* stack. Fortunately this cannot be the case, because otherwise property 3 of WRNF (we just use the same numbering of properties as for WNFs) would have been violated already in the original state.

- In rule 2, we have that both t_1 and t_2 are WNFs because $t_1 \cdot t_2$ is a WNF. Therefore J3 and J5 hold. The other invariants hold trivially.

- In rule 3 only invariant J4 needs special consideration. We have to show that t is a SNF, i.e. t contains no closures. We already assume because of J3 that t must be a WNF. That means it cannot be a λ-term or a *Computed* term. It also cannot be an application, because otherwise rule 2 of strong reduction would have applied instead of rule 3. It also cannot be a *Closure*: because t is a WNF, it could only be a closure of shape $\mathit{Closure} \ldots (\lambda \ldots)$, but closures of this shape would have been handled by rule 1 instead of rule 3. Therefore t must be a variable or a constant which both contain no closures.Therefore J4 holds.

- In rule 4 all invariants obviously hold.
- In rule 5, invariant J3 holds because of J5, and J6 because of J4.
- In rule 6, invariant J4 holds because of J4 and J6.
- In rule 7 all invariants obviously hold. Note that s must be *SEmpty* and that rule 7 is always the last rule that is applied during strong reduction.

We have proven that all invariants J1 to J6 hold throughout strong reduction. Any terminal result (SEmpty, t) of strong reduction fulfills in particular J4 and J2. This implies that t is both a SNF and a WRNF. Because it is a SNF, it contains no closures anymore. Because it is a WRNF, it does not contain any β-redexes or *Computed* terms anymore, because in a WRNF these constructs survive only within closures. □

In Section 2.4.2 we described how to delay immediate evaluation of the last arguments of a function by moving them from the left hand side of a rule to the right hand side. This is directly supported by the Barras machine; actually, it is at the heart of its execution model. In order to evaluate $\mathit{Closure}\ E\ (t_1 \cdot t_2)$, t_2 is wrapped up in a closure and put into the stack in unevaluated form (weak reduction rule 1). Then $\mathit{Closure}\ E\ t_1$ is evaluated. The result of this evaluation might be an abstraction which does not reference its argument. In this case, t_2 will never be evaluated. One can summarize the evaluation strategy of the Barras machine as follows: β reduction computes its argument lazily, application of a program rule computes all of the arguments strictly.

2.5.3 The SML Machine

The *SML machine* is currently the fastest implementation of the abstract machine interface. This is achieved by translating an abstract machine program directly to Standard ML code. The SML machine is not as general as the Barras machine; abstract machine terms of shape $\lambda\ t$ are translated to SML functions, which can only be applied, but not inspected for their body t. Therefore if the result of the computation still contains SML functions, it cannot be translated back to an abstract machine term. This restriction could be lifted by a more complex translation from λ terms to SML functions which also wires some form of term management into the SML functions. But our experiments seem to indicate that this irrevocably leads to a severe degradation of computing performance by a factor of 5 to 10. Therefore, in order to achieve maximal speed, the generality of the SML machine has been sacrificed. Note that this is not as big a restriction as it may sound:

- When compared to the Barras machine, this just means that the SML machine does not perform strong reduction but only weak reduction.
- Constants may represent functions; only the ability to return *anonymous* functions is lost.
- Anonymous functions can be used *during* the computation; only at the end they need to be gone.

An SML program basically consists of data structures and the functions that operate on them. Earlier we showed an example of how computing with numerals can be encoded in SML (fig. 2.8). The data structure there is a `datatype` made up of the four constructors `Zero`, `Neg1`, `B0` and `B1`. The functions are those for addition `add`, multiplication `mult` and so on. We would like the SML machine to produce similar code for the abstract machine program consisting of the rules in Figures 2.4-2.7. There seems to be a straightforward enough recipe; divide all constants into two kinds, the function constants, and the data constants. The function constants are those constants which appear as head symbols on the left hand side of a program rule, e.g. *neg*, as it appears as a head symbol in the rule *neg Zero = Zero*. The data constants are those constants appearing in the rules which are *not* function constants, e.g. *Zero*. Then generate a `datatype` definition containing all data constants as constructors, and one function definition for each function constant.

This simple recipe would actually be an adequate one if we were generating SML code for Figures 2.4-2.6 only. But the rules in Figure 2.7 complicate things, because now both B_0 and B_1 are function constants and therefore no longer data constants. This cripples our `datatype`. There is also the problem of the `norm` function in Figure 2.8. Its origin is in the rules of Figure 2.7, obviously, but how exactly is the SML function `norm` derived from those rules?

Figuring out stuff like this is nothing an automatic translation tool is very good at. Fortunately, we can modify our simple recipe to get another simple recipe. Who says that the sets of function constants and data constants must be disjunct? We just change our definition of data constants: *any* constant appearing in the rules is a data constant. Therefore function constants are special data constants. This means that B_0 and B_1 both have two SML incarnations. They are translated into SML constructors `B0` and `B1`. *And* they are translated into SML functions `b0` and `b1`:

```
fun b0 Zero = Zero                fun b1 Neg1 = Neg1
  | b0 x    = B0 x                  | b1 x    = B1 x
```

Definition 2.20 ► *Data and Function Constants*

The set $\mathcal{D} \subset \mathbb{Z}$ is the set of data constants*, i.e. Const c appears in any of the rules of the abstract machine program iff $c \in \mathcal{D}$.*

The set $\mathcal{F} \subset \mathcal{D}$ is the set of function constants*, i.e. $c \in \mathcal{F}$ iff there is a rule $r = ([g_1,\ldots,g_n], p, t)$ of the abstract machine program and p has shape PConst c $[p_1,\ldots,p_m]$. We say then that r* belongs *to the function constant c.*

We assume that the input of the SML machine consists not only of the abstract machine program but also of an arity function ϕ. There is no restriction on ϕ except that it be a function from $\mathbb{Z}$ to $\mathbb{N}$. The arity ϕ is critical to the behavior of the SML machine. Nevertheless, the SML machine is required to implement the abstract machine interface, no matter what ϕ might be.

A pattern *PConst* c $[p_1,\ldots,p_n]$ is called *compatible* with the arity function ϕ if $\phi(c) = n$ holds and if all p_i for $i = 1,\ldots,m$ are compatible with ϕ. We modify the given abstract machine program in the following way:

1. Any rule $([g_1,\ldots,g_m]$, *PConst* c $[p_1,\ldots p_n], t)$ with $n > \phi(c)$ or any incompatible p_i is removed.

2. Each rule $([(a_1,b_1),\ldots,(a_m,b_m)]$, *PConst* c $[p_1,\ldots,p_n], t)$ such that all p_i are compatible and $\delta = \phi(c) - n \geq 0$ is replaced by the rule

$$([(a_1 \uparrow^{\delta-1}, b_1 \uparrow^{\delta-1}),\ldots,(a_m \uparrow^{\delta-1}, b_m \uparrow^{\delta-1})],\ \mathit{PConst}\ c\ [p_1,\ldots,p_n,\underbrace{\mathit{PVar},\ldots,\mathit{PVar}}_{\delta\text{ times}}],\ \tilde{t}\,)$$

where $(\dots(t\uparrow^{\delta-1}\cdot Var\ (\delta-1))\cdot\dots\cdot Var\ 0)\rightarrow_\beta^* \tilde{t}$ and there is no t' with $\tilde{t}\rightarrow_\beta t'$. [3]
We define $\uparrow^{-1}$ to be just the identity operator.
We also say that c has *at most lazy index* δ, and *at least strict index* n.

Note that if $a \Rightarrow_\tau b$ with respect to the modified program then also $a \Rightarrow_\tau b$ with respect to the original program. We can therefore forget the original program and work with the modified program but still claim that the SML machine implements the abstract machine interface correctly with respect to the original program as long as it implements the interface correctly with respect to the modified one. From now on we assume that only compatible patterns occur in the abstract machine program. With every function constant $c \in \mathcal{F}$ we associate its *lazy index* $lazy(c)$, which is the least i such that c has at most lazy index i, and its *strict index* $strict(c)$, which is the greatest i such that c has at least strict index i. Lazy and strict index of c sum up to

$$\phi(c) = strict(c) + lazy(c).$$

◄ Definition 2.21 *Induced SML Data Structure*

Let $\mathcal{D} = \{c_1,\dots,c_n\}$ be the set of size n of data constants. The induced SML data structure `Term` *is then defined by*

$$\begin{array}{lcl} \texttt{datatype Term} & = & \texttt{App of Term * Term} \\ & | & \texttt{Abs of (Term -> Term)} \\ & | & \texttt{Const of int} \\ & | & \Phi(c_1) \\ & \vdots & \\ & | & \Phi(c_n) \end{array}$$

where

$$\Phi(c) \quad = \quad \begin{cases} \texttt{C}c & \text{if } \phi(c) = 0 \\ \texttt{C}c\ \texttt{of}\ \underbrace{\texttt{Term * ... * Term}}_{\phi(c)\text{ times}} & \text{if } \phi(c) > 0 \end{cases}$$

◄ Example 2.1

Assume $\mathcal{D} = \{0, 3, 8\}$, and $\phi(0) = 4$, $\phi(3) = 0$, $\phi(8) = 1$. Then

$$\begin{array}{lcl} \Phi(0) & = & \texttt{C0 of Term * Term * Term * Term} \\ \Phi(3) & = & \texttt{C3} \\ \Phi(8) & = & \texttt{C8 of Term} \end{array}$$

This is the induced SML data structure:

```
datatype Term  =  App of Term * Term
               |  Abs of (Term -> Term)
               |  Const of int
               |  C0 of Term * Term * Term * Term
               |  C3
               |  C8 of Term
```

The SML equality operator = is not defined for values of type `Term` because in order to compare `Abs f` with `Abs g` one would have to compare the two functions `f` and `g`. But in order to cope with guarded rules we need to be able to compare two values of type `Term` for equality. Of course, this comparison will only be approximate, i.e. it might return `false` for values that will behave identical in all situations.

Definition 2.22 ► *Equality for* `Term`

$$
\begin{array}{lcl}
\texttt{eq (App (a1,a2)) (App (b1,b2))} & = & \texttt{eq a1 b1 andalso eq a2 b2} \\
\texttt{eq (Abs u) (Abs v)} & = & \texttt{false} \\
\texttt{eq (Const c1) (Const c2)} & = & \texttt{(c1 = c2)} \\
\texttt{eq (C}c_1\texttt{(u}_1\texttt{,...,u}_{\phi(c_1)}\texttt{)) (C}c_1\texttt{(v}_1\texttt{,...,v}_{\phi(c_1)}\texttt{))} & = & \texttt{eq u}_1\ \texttt{v}_1\ \texttt{andalso} \ldots \texttt{andalso eq u}_{\phi(c_1)}\ \texttt{v}_{\phi(c_1)} \\
 & \vdots & \\
\texttt{eq (C}c_n\texttt{(u}_1\texttt{,...,u}_{\phi(c_n)}\texttt{)) (C}c_n\texttt{(v}_1\texttt{,...,v}_{\phi(c_n)}\texttt{))} & = & \texttt{eq u}_1\ \texttt{v}_1\ \texttt{andalso} \ldots \texttt{andalso eq u}_{\phi(c_n)}\ \texttt{v}_{\phi(c_n)} \\
\texttt{eq u v} & = & \texttt{false}
\end{array}
$$

An abstract machine pattern p can be translated into a piece of SML code $[p]^k_{\text{SML}}$. Like in the translation from p to $[p]_T$ we number the free variables of the pattern from right to left starting with 0.

Definition 2.23 ► *Translation of Abstract Machine Patterns to SML*

$$
\begin{array}{lcl}
[\mathit{PConst}\ c\ []]^k_{\text{SML}} & = & [c]^k_{\text{SML}}\ \texttt{()} \\
[\mathit{PConst}\ c\ [p_1,\ldots,p_n]]^k_{\text{SML}} & = & [c]^k_{\text{SML}}\ q_1 \ldots q_n \\
\multicolumn{3}{l}{\quad \text{for } n > 0 \text{ where } q_j = \texttt{(p}_j\ \texttt{as}\ [p_j]_{\text{SML},I(j)}\texttt{)} \text{ and } I(j) = \sum_{k=j+1}^{n} |p_k|} \\
[c]^k_{\text{SML}} & = & \begin{cases} \texttt{c}c & \text{if } k = 0 \\ \texttt{c}c\texttt{'}k & \text{if } k > 0 \end{cases} \\
[\mathit{PVar}]_{\text{SML},i} & = & \texttt{x}i \\
[\mathit{PConst}\ c\ []]_{\text{SML},i} & = & \texttt{C}c \\
[\mathit{PConst}\ c\ [p_1,\ldots,p_n]]_{\text{SML},i} & = & \texttt{C}c\ \texttt{(}q_1\texttt{, ... ,}q_n\texttt{)} \\
\multicolumn{3}{l}{\quad \text{for } n > 0 \text{ where } q_j = [p_j]_{\text{SML},I(j)} \text{ and } I(j) = i + \sum_{k=j+1}^{n} |p_k|}
\end{array}
$$

Here are several examples of the translation from patterns to SML code:

Example 2.2 ►

$$
\begin{array}{lcl}
[\mathit{PConst}\ 8\ [\mathit{PVar}]]^0_{\text{SML}} & = & \texttt{c8 (p1 as x0)} \\
[\mathit{PConst}\ 8\ [\mathit{PVar}]]^3_{\text{SML}} & = & \texttt{c8'3 (p1 as x0)} \\
[\mathit{PConst}\ 3\ []]^0_{\text{SML}} & = & \texttt{c3 ()} \\
[\mathit{PConst}\ 3\ []]^6_{\text{SML}} & = & \texttt{c3'6 ()}
\end{array}
$$

$[\mathit{PConst}\ 0\ [\mathit{PVar},\ \mathit{PConst}\ 3\ [],\ \mathit{PVar},\ \mathit{PConst}\ 0\ [\mathit{PVar}, \mathit{PConst}\ 8\ [\mathit{PVar}], \mathit{PVar}, \mathit{PVar}]]]^{13}_{\text{SML}} =$

```
c0'13 (p1 as x5) (p2 as C3) (p3 as x4) (p4 as C0 (x3, C8 (x2), x1, x0))
```

Also, any abstract machine term can be translated into a piece of SML code. For a more concise description of the translation we introduce a new constructor *Call* for abstract machine terms which can also be viewed as an abbreviation

$$\mathit{Call}\ c\ [t_1,\ldots,t_n] \quad := \quad (\ldots((\mathit{Const}\ c)\cdot t_1)\cdot \ldots \cdot t_n)$$

Definition 2.24 ► *Introducing Call*

1.	$\mathit{Const}\ c \to_{\text{call intro}} \mathit{Call}\ c\ []$	if	$c \in \mathcal{D}$
2.	$(\mathit{Call}\ c\ [a_1,\ldots,a_n])\cdot b \to_{\text{call intro}} \mathit{Call}\ c\ [a_1,\ldots,a_n,b]$	if	$n < \phi(c)$
3.	$\lambda\ t \to_{\text{call intro}} \lambda\ t'$	if	$t \to_{\text{call intro}} t'$
4.	$t_1 \cdot t_2 \to_{\text{call intro}} t_1' \cdot t_2$	if	$t_1 \to_{\text{call intro}} t_1'$
5.	$t_1 \cdot t_2 \to_{\text{call intro}} t_1 \cdot t_2'$	if	$t_2 \to_{\text{call intro}} t_2'$
6.	$\mathit{Computed}\ t \to_{\text{call intro}} \mathit{Computed}\ t'$	if	$t \to_{\text{call intro}} t'$
7.	$\mathit{Call}\ c\ [\ldots,a_i,\ldots] \to_{\text{call intro}} \mathit{Call}\ c\ [\ldots,a_i',\ldots]$	if	$a_i \to_{\text{call intro}} a_i'$

[3] In our setting there is always such a t' because t can be simply typed. Otherwise, just say that the SML machine fails on any input.

For our translation to always work and type check in SML, calls to function or data constants $c \in \mathcal{D}$ without supplying all of their arguments $\phi(c)$ must be prohibited. This can be achieved by performing η abstraction where necessary.

Definition 2.25 *η Abstraction for Call*

1. $\mathit{Call}\ c\ [a_1, \ldots, a_n] \rightarrow_{\text{abstract}} \underbrace{\lambda \ldots \lambda}_{\delta \text{ times}} (\mathit{Call}\ c\ [a_1 \uparrow^{\delta-1}, \ldots, a_n \uparrow^{\delta-1}, \mathit{Var}\,(\delta-1), \ldots, \mathit{Var}\,0])$ if $\delta = \phi(c) - n > 0$
2. $\lambda\, t \rightarrow_{\text{abstract}} \lambda\, t'$ if $t \rightarrow_{\text{abstract}} t'$
3. $t_1 \cdot t_2 \rightarrow_{\text{abstract}} t_1' \cdot t_2$ if $t_1 \rightarrow_{\text{abstract}} t_1'$
4. $t_1 \cdot t_2 \rightarrow_{\text{abstract}} t_1 \cdot t_2'$ if $t_2 \rightarrow_{\text{abstract}} t_2'$
5. $\mathit{Computed}\ t \rightarrow_{\text{abstract}} \mathit{Computed}\ t'$ if $t \rightarrow_{\text{abstract}} t'$
6. $\mathit{Call}\ c\ [\ldots, a_i, \ldots] \rightarrow_{\text{abstract}} \mathit{Call}\ c\ [\ldots, a_i', \ldots]$ if $a_i \rightarrow_{\text{abstract}} a_i'$

Given an abstract machine term t, let $[t]_{\text{call intro}}$ be the normal form with respect to $\rightarrow_{\text{call intro}}$, and $[t]_{\text{abstract}}$ the normal form with respect to $\rightarrow_{\text{abstract}}$. For the translation from an abstract machine term t to its corresponding SML code $[t]^l_{\text{SML}}$ we reuse the notation that we have employed for the translation from patterns to SML code; note that the indices in the notation have a different meaning, though.

Definition 2.26 *Translation of Abstract Machine Terms to SML*

$$[t]^l_{\text{SML}} = [[[t]_{\text{call intro}}]_{\text{abstract}}]^l_{\text{SML},0}$$

$$[\mathit{Const}\ c]^l_{\text{SML},m} = \texttt{(Const}\ c\texttt{)}$$

$$[\mathit{Var}\ v]^l_{\text{SML},m} = \begin{cases} \texttt{(b}(m-v-1)\texttt{)} & \text{if } v < m \\ \texttt{(x}(v-m)\ \texttt{())} & \text{if } m \le v < m+l \\ \texttt{(x}(v-m)\texttt{)} & \text{if } m+l \le v \end{cases}$$

$$[u \cdot v]^l_{\text{SML},m} = \texttt{(app}\ [u]^l_{\text{SML},m}\ [v]^l_{\text{SML},m}\texttt{)}$$

$$[\lambda\, t]^l_{\text{SML},m} = \texttt{(Abs (fn b}m\ \texttt{=>}\ [t]^l_{\text{SML},m+1}\texttt{))}$$

$$[\mathit{Computed}\ t]^l_{\text{SML},m} = [t]^l_{\text{SML},m}$$

$$[\mathit{Call}\ c\ [t_1, \ldots, t_n]]^l_{\text{SML},m} = \begin{cases} \texttt{C}c & \text{if } n = 0 \text{ and } c \notin \mathcal{F} \\ \texttt{(c}c\ \texttt{())} & \text{if } n = 0 \text{ and } c \in \mathcal{F} \\ \texttt{(C}c\ \texttt{(}[t_1]^l_{\text{SML},m}\texttt{,} \ldots \texttt{,} [t_n]^l_{\text{SML},m}\texttt{))} & \text{if } n > 0 \text{ and } c \notin \mathcal{F} \end{cases}$$

$$[\mathit{Call}\ c\ [t_1, \ldots, t_n]]^l_{\text{SML},m} = \texttt{(c}c\ [t_1]^l_{\text{SML},m} \cdots [t_{\mathit{strict}(c)}]^l_{\text{SML},m}\ \texttt{(fn () =>}\ [t_{\mathit{strict}(c)+1}]^l_{\text{SML},m}\texttt{)} \cdots \texttt{(fn () =>}\ [t_{\mathit{strict}(c)+\mathit{lazy}(c)}]^l_{\text{SML},m}\texttt{))}$$
if $n > 0$ and $c \in \mathcal{F}$

In the above we use the SML function `app`. Its definition is

```
fun app (Abs a) b   =   a b
  | app a b         =   App (a, b)
```
(2.5)

It is now time to deal with the question of how to translate the rules of the abstract machine program into SML functions. All rules which belong to the same function constant are grouped together. There are $|\mathcal{F}|$ such groups. Each group is converted to a bunch of mutual recursive SML functions. Actually, the generated SML functions of two different groups are also potentially mutually recursive. Therefore we specify for each abstract machine rule just an SML rule of the form $f\ p_1\ \ldots\ p_n = g$. The list of all such SML rules can then be converted into a list of mutual recursive SML function definitions by putting `fun`, `|` or `and` as appropiate in front of each SML rule.

Let $G_c = [r_1, \ldots, r_n]$ be the list of all rules of the abstract machine program which belong to the function constant $c \in \mathcal{F}$. With each $i \in \{1, \ldots, n+1\}$ we associate an index k_i in the following way:

- We set $k_1 = 0$.
- For $i > 1$ we set $k_i = k_{i-1}$ if r_{i-1} has no guards, and $k_i = k_{i-1} + 1$ otherwise.

The group G_c is converted into $|\{k_1, \ldots, k_{n+1}\}|$ mutual recursive SML functions which together consist of $n + |\{k_1, \ldots, k_{n+1}\}|$ SML rules. Each rule

$$r_i \quad = \quad ([(a_1, b_1), \ldots, (a_m, b_m)],\ p,\ t)$$

is converted either for $m = 0$ into the SML rule

$$[p]_{\mathtt{SML}}^{k_i} \ = \ [t]_{\mathtt{SML}}^{\mathtt{lazy}(c)}$$

or for $m > 0$ into the SML rule

$$\begin{array}{lcl}[p]_{\mathtt{SML}}^{k_i} & = & \mathtt{if\ eq}\ [a_1]_{\mathtt{SML}}^{\mathtt{lazy}(c)}\ [b_1]_{\mathtt{SML}}^{\mathtt{lazy}(c)}\ \mathtt{andalso}\ \ldots\ \mathtt{andalso\ eq}\ [a_m]_{\mathtt{SML}}^{\mathtt{lazy}(c)}\ [b_m]_{\mathtt{SML}}^{\mathtt{lazy}(c)} \\ & & \mathtt{then}\ [t]_{\mathtt{SML}}^{\mathtt{lazy}(c)}\ \mathtt{else}\ [c]_{\mathtt{SML}}^{k_{n+1}}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_{\phi(c)}\end{array}$$

This gives us n SML rules for the function constant c.

For each $k \in \{k_1, \ldots, k_{n+1}\}$ we have a rule which is triggered when the SML function $[c]_{\mathrm{SML}} k$ is applied to arguments for which there is no rule in the abstract machine program. For $\mathit{strict}(c) = \phi(c) > 0$ it is

$$[c]_{\mathtt{SML}}^{k}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_{\phi(c)} \ = \ \mathtt{C}c\ \ (\mathtt{p}_1, \ldots, \mathtt{p}_{\phi(c)})$$

For $\phi(c) = 0$ the default rule is

$$[c]_{\mathtt{SML}}^{k}\ \ \mathtt{()} \ = \ \mathtt{C}c$$

For $\mathit{lazy}(c) > 0$ it is

$$[c]_{\mathtt{SML}}^{k}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_{\phi(c)} \ = \ \mathtt{C}c\ \ (\mathtt{p}_1, \ldots, \mathtt{p}_{\mathtt{strict}(c)}, \mathtt{p}_{\mathtt{strict}(c)+1}\mathtt{()}, \ldots, \mathtt{p}_{\mathtt{strict}(c)+\mathtt{lazy}(c)}\mathtt{()})$$

Actually, experience shows that this last case seems to be rather useless, because it evaluates *all* lazy arguments of a function and will therefore most likely lead to nontermination. When the default rule is triggered this normally indicates that the abstract machine program is missing additional rules for evaluating the strict arguments of the function. An alternative and more user-friendly default case for $\mathit{lazy}(c) > 0$ is

$$[c]_{\mathtt{SML}}^{k}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_{\phi(c)} \ = \ \mathtt{raise\ UnresolvedLazyCall}$$

The semantics is usually the same, but in the second case the user will be informed of the failure of the SML machine not by nontermination, but by an exception.

We are almost there now. But before we can define the SML machine, we need to define how to convert an SML value t of type `Term` back into an abstract machine term $[t]_{\mathrm{AMT}}$. This is done in the obvious way:

Definition 2.27 ▸ *Translation of SML Values of Type `Term` to Abstract Machine Terms*

$$\begin{array}{lcl}[\mathtt{App}\ \ (a, b)]_{\mathrm{AMT}} & = & [a]_{\mathrm{AMT}} \cdot [b]_{\mathrm{AMT}} \\ [\mathtt{Abs}\ \ a]_{\mathrm{AMT}} & = & \text{undefined} \\ [\mathtt{Const}\ c]_{\mathrm{AMT}} & = & \mathit{Const}\, c \\ [\mathtt{C}c\ \ (t_1, \ldots, t_n)]_{\mathrm{AMT}} & = & (\ldots((\mathit{Const}\, c) \cdot [t_1]_{\mathrm{AMT}}) \cdot \ldots) \cdot [t_n]_{\mathrm{AMT}}\end{array}$$

The translation is a partial function as it works only on SML values which do not contain `Abs` nodes.

Definition 2.28 *SML Machine*

Let S be the SML program consisting of

- *the* `Term` *data type definition given in Definition 2.21,*
- *the SML function* `eq` *defined in Definition 2.22,*
- *the SML function* `app` *defined in Equation 2.5,*
- *the mutual recursive SML functions made up of all the SML rules stemming from converting each group G_c for all $c \in \mathcal{F}$.*

The SML machine *maps an arity function ϕ and an abstract machine program to the relation $\rightarrow_{SML}$ where*

$$t \rightarrow_{SML} t'$$

if

1. *t is a closed abstract machine term,*
2. *the result of executing $[t]^0_{SML}$ in the context of S is the SML value s,*
3. *s does not contain any occurrences of* `Abs`*,*
4. *$t' = [s]_{AMT}$.*

Example 2.3

Let us look at the power of a function. We can formalize this concept in Isabelle/HOL by defining a constant *power* which obeys the following equation:

$$\textit{power}\ f\ n = \textbf{if}\ \textit{test-le}\ n\ \textbf{then}\ \lambda\ x.\ x\ \textbf{else}\ \lambda\ x.\ \textit{power}\ f\ (\textit{add}\ n\ \textit{Neg}_1)\ (f\ x) \tag{2.6}$$

Alternatively, we could use the following equation:

$$\textit{power}\ f\ n\ x = \textbf{if}\ \textit{test-le}\ n\ \textbf{then}\ x\ \textbf{else}\ \textit{power}\ f\ (\textit{add}\ n\ \textit{Neg}_1)\ (f\ x) \tag{2.7}$$

We could also describe *power* by the guarded equations

$$\begin{array}{lcccl} \textit{test-le}\ n & \equiv & \textit{True} & \Longrightarrow & \textit{power}\ f\ n\ x = x \\ \textit{test-le}\ n & \equiv & \textit{False} & \Longrightarrow & \textit{power}\ f\ n\ x = \textit{power}\ f\ (\textit{add}\ n\ \textit{Neg}_1)\ (f\ x) \end{array} \tag{2.8}$$

Based on *power* we define the square of a nonnegative integer in terms of addition:

$$\textit{square}\ a = \textit{power}\ (\textit{add}\ a)\ a\ \textit{Zero} \tag{2.9}$$

This defines a function *square* such that *square* $a = a^2$ for $a \geq 0$. The abstract machine program for executing *square* consists of the following rules:

- those for *add* and $\textit{add}_1$ (fig. 2.5);
- those for normalizing numerals (fig. 2.7);
- those for *test-le* and *test-less* (fig. 2.9);
- those for *If* (eq. 2.4);
- those for *power* (either eq. 2.6 or eq. 2.7 or eq. 2.8);
- those for *square* (eq. 2.9).

The set of function constants is then

c	$c \in \mathcal{F}$	$\phi(c)$	*strict*(c)	*lazy*(c)	
TRUE	no	0			
FALSE	no	0			
ZERO	no	0			
NEG1	no	0			
B0	yes	1	1	0	
B1	yes	1	1	0	
ADD	yes	2	2	0	
ADD1	yes	2	2	0	
TESTLE	yes	1	1	0	
TESTLESS	yes	1	1	0	
IF	yes	3	1	2	
POWER	yes	3	2	1	for Equation 2.6
			3	0	for Equation 2.7
			3	0	for Equation 2.8
SQUARE	yes	1	1	0	

Table 2.3: Arity, Strictness, Laziness for Elements of $\mathcal{D}$

$$\mathcal{F} = \{\texttt{B0, B1, ADD, ADD1, TESTLE, TESTLESS, IF, POWER, SQUARE}\}$$

and the set of data constants is $\mathcal{D} = \{\texttt{TRUE, FALSE, ZERO, NEG1}\} \cup \mathcal{F}$. For a more readable presentation, we use textual labels for the elements of $\mathcal{D}$. The assumed arity function ϕ and the derived functions *lazy* and *strict* are listed in Table 2.3. Note that *lazy*(POWER) and *strict*(POWER) depend on what choice we have made in selecting a rule for *power* from the three equations (2.6), (2.7) and (2.8). The resulting SML program S is displayed in Figures 2.11, 2.12 and 2.13, 2.14 or 2.15, respectively.

Let us compute $t = square\ (B_1\,(B_1\,Zero))$, which amounts to calculating 3^2. Executing $[t]^0_{\mathrm{SML}}$ = `cSQUARE (cB1 (cB1 CZERO))` in the context of S results in the SML value s = `CB1 (CB0 (CB0 (CB1 ZERO)))`. Translating s back into the realm of abstract machine terms gives $[s]_{\mathrm{AMT}} = B_1\,(B_0\,(B_0\,(B_1\,Zero)))$, which is the numeral representation of 9.

Let us also illustrate the restriction of the SML machine to fail on abstract machine terms which compute to terms still containing abstractions. Assuming we use choose Equation 2.7 or 2.8 as rule for *power*, translating $t = power\ (add\ Zero)\ Zero$ leads to

$[t]^0_{\mathrm{SML}}$ = `Abs (fn b0 => cPOWER (Abs (fn b1 => cADD CZERO b1)) CZERO b0)`

Executing $[t]^0_{\mathrm{SML}}$ in the context of S returns an SML value `Abs` f for some f of type `Term -> Term`. We do not have enough information to translate such an SML value back into an abstract machine term. On the other hand, executing $t = (\lambda\ x.power\ (add\ Zero)\ Zero\ x)\ Zero$ is no problem:

```
app (Abs (fn b0 => cPOWER (Abs (fn b1 => cADD CZERO b1)) CZERO b0)) CZERO
```

computes to the SML value `CZERO`, and the corresponding abstract machine term is *Zero*.

Theorem 2.2 ► *Partial Correctness of the SML Machine*

The SML machine is for any arity function ϕ an abstract machine in the sense of Definition 2.11.

Proof. We do not provide a rigorous proof of this theorem. Such a proof would involve a semantics for SML, and then a proof that τ reduction on abstract machine terms corresponds to evaluation of the translations in SML. Let us rather just note

```
datatype Term =
    App of Term * Term
  | Abs of Term -> Term
  | Const of int
  | CTRUE | CFALSE | CZERO | CNEG1
  | CB0 of Term | CB1 of Term
  | CADD of Term * Term | CADD1 of Term * Term
  | CTESTLE of Term | CTESTLESS of Term
  | CIF of Term * Term * Term
  | CPOWER of Term * Term * Term
  | CSQUARE of Term

fun eq (App (a1, a2)) (App (b1, b2)) = eq a1 b1 andalso eq a2 b2
  | eq (Abs u) (Abs v) = false
  | eq (Const c1) (Const c2) = (c1 = c2)
  | eq CTRUE CTRUE = true
  | eq CFALSE CFALSE = true
  | eq CZERO CZERO = true
  | eq CNEG1 CNEG1 = true
  | eq (CB0 u1) (CB0 v1) = eq u1 v1
  | eq (CB1 u1) (CB1 v1) = eq u1 v1
  | eq (CADD (u1,u2)) (CADD (v1,v2)) = eq u1 v1 andalso eq u2 v2
  | eq (CADD1 (u1,u2)) (CADD1 (v1,v2)) = eq u1 v1 andalso eq u2 v2
  | eq (CTESTLE u1) (CTESTLE v1) = eq u1 v1
  | eq (CTESTLESS u1) (CTESTLESS v1) = eq u1 v1
  | eq (CIF (u1,u2,u3)) (CIF (v1,v2,v3)) = eq u1 v1 andalso eq u2 v2 andalso eq u3 v3
  | eq (CPOWER (u1,u2,u3)) (CPOWER (v1,v2,v3)) = eq u1 v1 andalso eq u2 v2 andalso eq u3 v3
  | eq (CSQUARE u1) (CSQUARE v1) = eq u1 v1
  | eq u v = false

fun app (Abs a) b = a b
  | app a b = App (a, b)

fun cB0 CZERO = CZERO
  | cB0 p1 = CB0 p1

and cB1 (p1 as CNEG1) = CNEG1
  | cB1 p1 = CB1 p1

and cADD (p1 as (CB0 x1)) (p2 as (CB0 x0)) = cB0 (cADD x1 x0)
  | cADD (p1 as (CB0 x1)) (p2 as (CB1 x0)) = cB1 (cADD x1 x0)
  | cADD (p1 as (CB1 x1)) (p2 as (CB0 x0)) = cB1 (cADD x1 x0)
  | cADD (p1 as (CB1 x1)) (p2 as (CB1 x0)) = cB0 (cADD1 x1 x0)
  | cADD (p1 as CZERO) (p2 as x0) = x0
  | cADD (p1 as x0) (p2 as CZERO) = x0
  | cADD (p1 as CNEG1) (p2 as (CB0 x0)) = cB1 (cADD CNEG1 x0)
  | cADD (p1 as CNEG1) (p2 as (CB1 x0)) = cB0 x0
  | cADD (p1 as (CB0 x0)) (p2 as CNEG1) = cB1 (cADD x0 CNEG1)
  | cADD (p1 as (CB1 x0)) (p2 as CNEG1) = cB0 x0
  | cADD (p1 as CNEG1) (p2 as CNEG1) = cB0 CNEG1
  | cADD p1 p2 = CADD (p1, p2)
```

Figure 2.11: SML Program, Part 1

```
and cADD1 (p1 as (CB0 x1)) (p2 as (CB0 x0)) = cB1 (cADD x1 x0)
  | cADD1 (p1 as (CB0 x1)) (p2 as (CB1 x0)) = cB0 (cADD1 x1 x0)
  | cADD1 (p1 as (CB1 x1)) (p2 as (CB0 x0)) = cB0 (cADD1 x1 x0)
  | cADD1 (p1 as (CB1 x1)) (p2 as (CB1 x0)) = cB1 (cADD1 x1 x0)
  | cADD1 (p1 as CNEG1) (p2 as x0) = x0
  | cADD1 (p1 as x0) (p2 as CNEG1) = x0
  | cADD1 (p1 as CZERO) (p2 as (CB0 x0)) = cB1 x0
  | cADD1 (p1 as CZERO) (p2 as (CB1 x0)) = cB0 (cADD1 CZERO x0)
  | cADD1 (p1 as (CB0 x0)) (p2 as CZERO) = cB1 x0
  | cADD1 (p1 as (CB1 x0)) (p2 as CZERO) = cB0 (cADD1 x0 CZERO)
  | cADD1 (p1 as CZERO) (p2 as CZERO) = cB1 CZERO
  | cADD1 p1 p2 = CADD1 (p1, p2)

and cTESTLE (p1 as CZERO) = CTRUE
  | cTESTLE (p1 as CNEG1) = CTRUE
  | cTESTLE (p1 as (CB0 x0)) = cTESTLE x0
  | cTESTLE (p1 as (CB1 x0)) = cTESTLESS x0
  | cTESTLE p1 = CTESTLE p1

and cTESTLESS (p1 as CZERO) = CFALSE
  | cTESTLESS (p1 as CNEG1) = CTRUE
  | cTESTLESS (p1 as (CB0 x0)) = cTESTLESS x0
  | cTESTLESS (p1 as (CB1 x0)) = cTESTLESS x0
  | cTESTLESS p1 = CTESTLESS p1

and cIF (p1 as CTRUE) (p2 as x1) (p3 as x0) = x1 ()
  | cIF (p1 as CFALSE) (p2 as x1) (p3 as x0) = x0 ()
  | cIF p1 p2 p3 = raise UnresolvedLazyCall
```

Figure 2.12: SML Program, Part 2

```
and cPOWER (p1 as x2) (p2 as x1) (p3 as x0) =
      app (cIF (cTESTLE x1)
               (fn () => Abs (fn b0 => b0))
               (fn () => Abs (fn b0 => cPOWER x2 (cADD x1 CNEG1) (fn () => app x2 b0))))
          (x0 ())
  | cPOWER p1 p2 p3 = raise UnresolvedLazyCall

and cSQUARE (p1 as x0) = cPOWER (Abs (fn b0 => cADD x0 b0)) x0 (fn () => CZERO)
  | cSQUARE p1 = CSQUARE p1
```

Figure 2.13: SML Program, Part 3a (resulting from Equation 2.6)

```
and cPOWER (p1 as x2) (p2 as x1) (p3 as x0) =
      cIF (cTESTLE x1) (fn () => x0) (fn () => cPOWER x2 (cADD x1 CNEG1) (app x2 x0))
  | cPOWER p1 p2 p3 = CPOWER (p1, p2, p3)

and cSQUARE (p1 as x0) = cPOWER (Abs (fn b0 => cADD x0 b0)) x0 CZERO
  | cSQUARE p1 = CSQUARE p1
```

Figure 2.14: SML Program, Part 3b (resulting from Equation 2.7)

```
and cPOWER (p1 as x2) (p2 as x1) (p3 as x0) =
      if eq (cTESTLE x1) CTRUE then x0
      else cPOWER'1 p1 p2 p3
  | cPOWER p1 p2 p3 = CPOWER (p1, p2, p3)

and cPOWER'1 (p1 as x2) (p2 as x1) (p3 as x0) =
      if eq (cTESTLE x1) CFALSE then cPOWER x2 (cADD x1 CNEG1) (app x2 x0)
      else cPOWER'2 p1 p2 p3
  | cPOWER'1 p1 p2 p3 = CPOWER (p1, p2, p3)

and cPOWER'2 p1 p2 p3 = CPOWER (p1, p2, p3)

and cSQUARE (p1 as x0) = cPOWER (Abs (fn b0 => cADD x0 b0)) x0 CZERO
  | cSQUARE p1 = CSQUARE p1
```

Figure 2.15: SML Program, Part 3c (resulting from Equation 2.8)

that we use SML as a purely functional language without side effects. Moving from the left hand side of a generated SML function definition to the right hand side of it and evaluating then this right hand side can clearly be understood as τ reduction. Also, all massaging of abstract machine terms and patterns we perform when ensuring that the abstract machine program is compatible with ϕ, and when translating t to $[t]^l_{\mathrm{SML}}$, can also be understood in terms of τ reduction. □

2.5.4 The Haskell Machine

The Haskell machine is just a simpler version of the SML machine. Because Haskell is a lazy language, there is no need to distinguish between lazy and strict arguments of a function; we can just treat all function arguments like we treated the strict function arguments. Apart from that we could just copy everything said in the previous subsection about the SML machine to this subsection with minor modifications due to the syntactic differences between Haskell and SML. If we were willing to invest a little more work, we could further simplify the translation from abstract machine terms to Haskell code by utilizing that Haskell already has a built-in concept of guards, and that Haskell allows functions of arity 0.

2.6 The HCL Cokernel

The *HCL cokernel* provides secure access to the modes of the HCL. It can be understood as an additional kernel sitting beside the Isabelle kernel and we will discuss its facilities for producing theorems. We argue that the cokernel cannot produce any theorem that the Isabelle kernel could not also produce on its own. This is what it means for the cokernel to be secure.

2.6.1 A Bird's-Eye View of the Isabelle Kernel

The Isabelle kernel is built around the following main notions: *types*, *terms*, *theorems* and *theories*.

A *theory* $\mathfrak{T}$ can be viewed as a state of the kernel. This state has (among others) the following components:

- A set of *axiomatic type classes*, like *ring* or *order*.
- A set of *type constructors*, like *real* or *prop* or $\rightarrow$. Each type constructor has an associated arity. E.g. *real* and *prop* have both arity 0, and $\rightarrow$ has arity 2. The type constructors *prop* and $\rightarrow$ are part of *every* Isabelle theory.

A *sort* is a subset of the set of axiomatic type classes. *Types* are given via

$$type ::= TVar\ \alpha\ S \mid TApp\ c\ (type_1, \ldots, type_n)$$

where α is some identifier, S is a sort, and c is a type constructor with associated arity n. Instead of $TApp \rightarrow (\tau_1, \tau_2)$ we also write $\tau_1 \rightarrow \tau_2$. Instead of $TApp\ c$ where c has arity 0 we just write c. We call $TVar\ \alpha\ S$ a *type variable* and often write $\alpha :: S$ for it. We call a type τ a *ground type* if it does not contain any type variable. We define the *functional arity* of a type τ to be 0 if τ is a type variable or not of shape $\tau_1 \rightarrow \tau_2$, and to be $1+$ the functional arity of τ_2 if it has shape $\tau_1 \rightarrow \tau_2$.

There are two more components of the theory $\mathfrak{T}$:

- A transitive instance relation $\leq$ on types such that $\tau_1 \leq \tau_2$ if τ_1 can be obtained from τ_2 by instantiating type variables $TVar\ \alpha\ S$. For example, in many theories we will have $real \leq \alpha :: \{ring\}$ and $real \rightarrow nat \leq \alpha :: \{ring\} \rightarrow \beta :: \{order\}$, but not $nat \leq \alpha :: \{ring\}$. As always, when instantiating a type variable it must be instantiated *everywhere* in the type, therefore we can never have $real \rightarrow \alpha :: \{ring\} \leq \alpha :: \{ring\} \rightarrow \alpha :: \{ring\}$. We have[4] $\tau \leq \alpha :: \{\}$ for any type τ. We just write α for $\alpha :: \{\}$.
- A set of *constants*. Each constant is associated with its most general type τ. Equality $\equiv$ with most general type $\alpha \rightarrow \alpha \rightarrow prop$ and implication $\Longrightarrow$ with most general type $prop \rightarrow prop$ are both constants that are part of *every* Isabelle theory.

Isabelle terms use de-Bruijn indices [6] for local variables and names for free variables. There are two kinds of free variables, *Free* and *Var*.

$$\begin{array}{lcl} term & ::= & App\ term_1\ term_2 \\ & \mid & Const\ c\ type \\ & \mid & Var\ x\ type \\ & \mid & Free\ x\ type \\ & \mid & \lambda\ type.\ term \\ & \mid & Bound\ i \end{array}$$

In the above c is a constant of the theory, x is an identifier, and i is a de-Bruijn index. We call a term *welltyped* if it can be assigned a type using the following typing rules:

Definition 2.29 ► *Type of a Term*

$$\begin{array}{ll}
\mathrm{type}(t) = \tau & \text{if } \mathrm{type}_0\ []\ t = \tau \\
\mathrm{type}_0\ E\ (Const\ c\ \tau) = \tau & \text{if } \tau \leq \tau_c \text{ where } \tau_c \text{ is the most general type of } c \\
\mathrm{type}_0\ E\ (App\ t_1\ t_2) = \tau & \text{if } \mathrm{type}_0\ E\ t_1 = \tau_2 \rightarrow \tau \text{ and } \mathrm{type}_0\ E\ t_2 = \tau_2 \\
\mathrm{type}_0\ E\ (Var\ x\ \tau) = \tau & \\
\mathrm{type}_0\ E\ (Free\ x\ \tau) = \tau & \\
\mathrm{type}_0\ [\tau_1, \ldots, \tau_n]\ (\lambda\ \tau.\ t) = \tau \rightarrow \tau' & \text{if } \mathrm{type}_0\ [\tau, \tau_1, \ldots, \tau_n]\ t = \tau' \\
\mathrm{type}_0\ [\tau_1, \ldots, \tau_n]\ (Bound\ i) = \tau_{i+1} & \text{if } i < n
\end{array}$$

[4] This is not exactly how it works in Isabelle, but good enough for our purposes.

When writing down terms we often use abbreviations whose meaning we take for obvious. For example, we will write $t_1 \equiv t_2$ instead of

$$App\ (App\ (Const\ \equiv\ (type(t_1) \to type(t_2)))\ t_1)\ t_2.$$

A *theorem* consists of the following components:

- Its *proposition*, which is a welltyped term of type *prop*.
- A set of *hypotheses*. A hypothesis is a welltyped term of type *prop* which does not contain any occurrences of *Var*.
- A set of *sort hypotheses*. A sort hypothesis is just a sort S. Its meaning is that it is assumed that there is a ground type τ such that $\tau \leq \alpha :: S$.

We write $t = (\mathcal{S},\ \mathcal{H},\ p)$ for a theorem t with sort hypotheses $\mathcal{S}$, hypotheses $\mathcal{H}$ and proposition p. While the Isabelle kernel allows to access the components of a theorem, one cannot construct a theorem by giving its components. Theorems can only be constructed through a set of operations the kernel provides. Here are a few of those operations:

Reflexivity Given a welltyped term t, the kernel can construct the theorem $(\mathcal{S}, \{\}, t \equiv t)$, where $\mathcal{S}$ is the set of all sorts occurring in t.

Transitivity of Equality Given two theorems $(\mathcal{S}_1,\ \mathcal{H}_1,\ t_1 \equiv t_2)$ and $(\mathcal{S}_2,\ \mathcal{H}_2,\ t_2 \equiv t_3)$, the kernel can construct the theorem $(\mathcal{S}_1 \cup \mathcal{S}_2,\ \mathcal{H}_1 \cup \mathcal{H}_2,\ t_1 \equiv t_3)$.

Discarding Sort Hypotheses Given a ground type τ, and a type variable $\alpha :: S$ such that $\tau \leq \alpha :: S$, and a theorem $(\mathcal{S},\ \mathcal{H},\ p)$, the kernel can construct the theorem $(\mathcal{S} - \{S\},\ \mathcal{H},\ p)$.

Instantiation of Type Variables Given different type variables $\alpha_1 :: S_1, \ldots, \alpha_n :: S_n$, and n types τ_i with $\tau_i \leq \alpha_i :: S_i$, and a theorem $(\mathcal{S},\ \mathcal{H},\ p)$, the kernel can construct the theorem $(\mathcal{S},\ \mathcal{H}',\ p')$, where $\mathcal{H}'$ originates from $\mathcal{H}$ and p' originates from p by simultaneously replacing all occurrences of $\alpha_i :: S_i$ by τ_i.

Instantiation of Variables Given different variables $TFree\ x_1\ \tau_1, \ldots, TFree\ x_n\ \tau_n$ and n terms t_i with $\text{type}(t_i) = \tau_i$, and a theorem $(\mathcal{S},\ \mathcal{H},\ p)$, the kernel can construct the theorem $(\mathcal{S},\ \mathcal{H}',\ p')$, where $\mathcal{H}'$ originates from $\mathcal{H}$ and p' originates from p by simultaneously substituting (this may involve de-Bruijn lifting of the t_i) all occurrences of $TFree\ x_i\ \tau_i$ by t_i. The same is true when we use in the above *TVar* instead of *TFree*.

$\beta\eta$ **reduction** Given a welltyped term t, and another welltyped term t' such that t' results from t by $\beta\eta$ reduction, the kernel can construct the theorem $(\mathcal{S},\ \{\},\ t \equiv t')$, where $\mathcal{S}$ is the set of all sorts occurring in t.

Modus Ponens Given two theorems $(\mathcal{S}_1,\ \mathcal{H}_1,\ a \Longrightarrow b)$ and $(\mathcal{S}_2,\ \mathcal{H}_2,\ a)$, the kernel can construct the theorem $(\mathcal{S}_1 \cup \mathcal{S}_2,\ \mathcal{H}_1 \cup \mathcal{H}_2,\ b)$.

2.6.2 Removing and Attaching Types

One of the reasons why computing using the Isabelle kernel is slow is that the basic datastructure for computing used by the Isabelle kernel is that of a theorem, which is made up of terms, which are stuffed with types not really needed for computing. These types must be manipulated during computation, and can grow awfully large. To avoid this performance pitfall, abstract machines compute on untyped terms. Therefore, in order to use any abstract machine for computation we need to bridge the gap between Isabelle terms, which are typed, and abstract machine terms.

So, how can we get rid of the types in Isabelle terms? Just dropping them like in going from $Const\ c\ \tau$ to $Const\ c$ is not an option. E.g., Isabelle allows the overloading of constants [34, 27]. An example where overloading comes in handy is when looking at integers *int* and nonnegative integers *nat*. The constants for subtraction $-$ and zero 0 can be used with both types. But the equations that these constants fulfill differ wildly. If 0 and 1 have type *nat*, then in Isabelle/HOL the equation $0 - 1 = 0$ holds. Applying this rewriting rule to $0 - 1$, where 0 and 1 have type *int*, is clearly desastrous.

There is an easy way out of these complications. The terms $Const\ 0\ int$ and $Const\ 0\ nat$ are different, so we map them to different abstract machine terms. The term $Const\ 0\ int$ could be mapped to $Const\ 14$, for example, and the term $Const\ 0\ nat$ to $Const\ 15$.

Definition 2.30 ▶ *Atoms*

We call an Isabelle term an atom *if it is either a constant $Const\ c\ \tau$ or a variable $Free\ x\ \tau$ or $Var\ x\ \tau$. For an arbitrary Isabelle term t we denote by* $\text{atoms}(t)$ *the set of all atoms occurring in t.*

Definition 2.31 ▶ *Encoding*

An encoding *σ is a partial injective mapping from the set of all welltyped atoms of a theory to $\mathbb{Z}$. Any encoding σ induces a total mapping $\overline{\sigma}$ from the set of all welltyped Isabelle terms t with* $\text{atoms}(t) \subseteq \text{dom}\,\sigma$ *to the set of closed pure abstract machine terms:*

$$\begin{array}{lcl}
\overline{\sigma}\,(Const\ c\ \tau) & = & Const\ (\sigma\,(Const\ c\ \tau)) \\
\overline{\sigma}\,(Var\ x\ \tau) & = & Const\ (\sigma\,(Var\ x\ \tau)) \\
\overline{\sigma}\,(Free\ x\ \tau) & = & Const\ (\sigma\,(Free\ x\ \tau)) \\
\overline{\sigma}\,(App\ u\ v) & = & (\overline{\sigma}\,u)\cdot(\overline{\sigma}\,v) \\
\overline{\sigma}\,(\lambda\ \tau.\ t) & = & \lambda\ (\overline{\sigma}\,t) \\
\overline{\sigma}\,(Bound\ i) & = & Var\ i
\end{array}$$

Let us assume now that we have an Isabelle term t we would like to compute. Given an encoding σ such that t is in the domain of $\overline{\sigma}$, we convert t to the abstract machine term $s = \overline{\sigma}\,t$. Computing s with an abstract machine AM results in another abstract machine term s' with $s \to_{\text{AM}} s'$. Therefore the result of computing t is $t' = \overline{\sigma}^{\,-1} s'$.

There is a flaw in this approach. The problem is that although σ is an injective mapping, $\overline{\sigma}$ is not. This can be seen quickly:

$$\overline{\sigma}\,(\lambda\ int.\ Bound\ 0) = \lambda\ (Var\ 0) = \overline{\sigma}\,(\lambda\ prop.\ Bound\ 0).$$

Fortunately, we know more about t' than just $\overline{\sigma}\,t' = s'$. We also know that any t' acceptable to us must have the property $\text{type}(t) = \text{type}(t')$. This is enough to find t'.

Theorem 2.3 ▶ *Reversibility of Encoding*

For any encoding σ, the mapping $t \mapsto (\overline{\sigma}\ t,\ \text{type}(t))$ is injective on the set of those terms in the domain of $\overline{\sigma}$ which contain no β redexes.

Proof. Let us first note that if t contains no β redexes, then neither does $\overline{\sigma}\,t$. Thus we can prove the theorem by defining a function *attach* which for a type τ and a closed pure abstract machine term s without β redexes either returns the unique t with $\text{type}(t) = \tau$ and $\overline{\sigma}\ t = s$ or signals failure if there is no such t. That t must be unique will be clear from the function definition itself because for each case of the function definition there is no degree of freedom what result could be returned.

$$\mathit{attach}\ \tau\ E\ (\mathit{Const}\ c) \quad = \quad \begin{cases} \sigma^{-1}\ c & \text{if } c \in \text{dom}\ \sigma \wedge \text{type}(\sigma^{-1}\ c) = \tau \\ \text{failure} & \text{otherwise} \end{cases}$$

$$\mathit{attach}\ \tau\ E\ (\mathit{Var}\ i) \quad = \quad \mathit{Bound}\ i$$

$$\mathit{attach}\ \tau\ [\tau_1,\ldots,\tau_n]\ (\lambda\ t) \quad = \quad \begin{cases} \lambda\ \tau'.\ \mathit{attach}\ \tau''\ [\tau',\tau_1,\ldots,\tau_n]\ t & \text{if } \tau = \tau' \to \tau'' \\ \text{failure} & \text{otherwise} \end{cases}$$

$$\mathit{attach}\ \tau\ E\ (f \cdot u_1 \cdot \ldots \cdot u_n)$$
$$= \begin{cases} \mathit{App} \ldots (\mathit{App}\ g\ v_1) \ldots v_n & \text{if } \mathit{attach}_0\ E\ f = (g, \tau_1 \to \ldots \to \tau_n \to \tau) \text{ and } v_i = \mathit{attach}\ \tau_i\ E\ u_i \\ \text{failure} & \text{otherwise} \end{cases}$$

$$\mathit{attach}_0\ E\ (\mathit{Const}\ c) \quad = \quad \begin{cases} (\sigma^{-1}\ c,\ \text{type}(\sigma^{-1}\ c)) & \text{if } c \in \text{dom}\ \sigma \\ \text{failure} & \text{otherwise} \end{cases}$$

$$\mathit{attach}_0\ [\tau_1,\ldots,\tau_n]\ (\mathit{Var}\ i) \quad = \quad (\mathit{Bound}\ i,\ \tau_{i+1})$$

Note that when defining *attach* for a chain of applications $f \cdot u_1 \cdot \ldots \cdot u_n$ we know that f cannot be an application $f = f_0 \cdot u_0$ because otherwise we would instead be looking at f_0 in the chain of applications $f_0 \cdot u_0 \cdot u_1 \cdot \ldots \cdot u_n$. Also, f cannot be an abstraction $\lambda\ t$ because then we would have found a β redex. Therefore f must either be a constant or a variable, and both of these cases are covered in the definition of attach_0. □

2.6.3 Computing Equations

A state of the Isabelle kernel is called a theory. The corresponding concept for the HCL cokernel is that of a *computer*. A computer contains all the information necessary for computing an Isabelle term. Given a computer $\mathfrak{C}$, and a welltyped Isabelle term t, the cokernel will compute t to t' and upon success return the equation $t \equiv t'$ as a theorem.

We create a computer $\mathfrak{C}$ by handing over to the cokernel the following items:

1. A theory $\mathfrak{T}$.
2. A mode AM, i.e. a tag AM $\in$ {BARRAS, SML, HASKELL} which indicates which abstract machine should be used for computing.
3. A list $[\Psi_1,\ldots,\Psi_n]$ of theorems of the theory $\mathfrak{T}$. The proposition p_i of each theorem

$$\Psi_i = (\mathcal{S}_i,\ \mathcal{H}_i,\ p_i)$$

   has the shape given in Equation 2.2.

The created computer $\mathfrak{C} = (\mathfrak{T},\ \text{AM},\ \mathcal{S},\ \mathcal{H},\ \sigma,\ \to_{\text{AM}})$ consists of these components:

- The theory $\mathfrak{T}$ and the mode AM.
- The cummulative set of sort hypotheses $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n \cup \mathcal{P}$ where $\mathcal{P}$ is the set of all sorts occurring in any of the p_i.
- The cummulative set of of hypotheses $\mathcal{H} = \mathcal{H}_1 \cup \ldots \cup \mathcal{H}_n$.
- An encoding σ induced by the set of propositions $\{p_1, \ldots, p_n\}$. Any encoding with $\text{atoms}(p_i) \subseteq \text{dom}\,\sigma$ for all $i = 1, \ldots, n$ will do.
- The partial abstract machine function $\rightarrow_{\text{AM}}$ induced by the arity function ϕ and the abstract machine program $[r_1, \ldots, r_n]$.

  There are several sensible choices for ϕ. One is to define $\phi(d)$ as the functional arity of $\text{type}(\sigma^{-1} d)$. Another one is to inspect the abstract machine program and make a heuristic choice which is our current approach.

  The abstract machine rule r_i is derived from the proposition p_i in the obvious way using the encoding σ. Note that we convert only *Var*s to pattern variables, but not *Free*s. Also converting *Free*s to pattern variables could lead to inconsistency as *Free*s may appear in the hypotheses $\mathcal{H}_i$, which is not the case for *Var*s.

The cokernel applies the computer $\mathfrak{C}$ to compute a welltyped term t of the theory $\mathcal{T}$ in the following way: [5]

1. Calculate $s = \overline{\sigma}\, t$.
2. Calculate s' such that $s \rightarrow_{\text{AM}} s'$.
3. Upon successfully calculating s', calculate $t' = \mathit{attach}\ (\mathit{type}(t))\ []\ s'$.
4. Upon successfully calculating t', return the theorem $\Psi = (\mathcal{S} \cup \mathcal{S}_t,\ \mathcal{H},\ t \equiv t')$, where $\mathcal{S}_t$ is the set of sorts occurring in t.

In order to show that the HCL cokernel is secure, we need to show that the theorem Ψ could also have been proven using only the operations the Isabelle kernel provides. We know that

$$\overline{\sigma}\, t \quad \Rightarrow_\tau \quad \overline{\sigma}\, t'$$

holds. Using the reflexivity rule of the Isabelle kernel we can start with the theorem $t \equiv t$ and then replay each τ reduction step using the Isabelle kernel operations, chaining them using the transitivity rule. For example, rule 3 of τ reduction in Definition 2.10 can be simulated using the instantiation of variables rule of the Isabelle kernel.

The only τ reduction step making trouble is η abstraction. For an untyped term s it is alright to go from s to $\lambda(s \uparrow^0 \cdot (\mathit{Var}\,0))$, but for a welltyped Isabelle term t such that $\delta = \text{type}(t)$ is not of shape $\tau_1 \rightarrow \tau_2$ this is not a valid step because it would lead to a term that is not welltyped any more. There is a workaround. We can just assume that everywhere in our proof so far the type δ has been replaced by the type $\mathit{unit} \rightarrow \delta$ and proceed replaying. We can do this because the type $\mathit{unit} \rightarrow \delta$ is isomorphic to δ, so for each constant c involving δ we could define a new constant

[5] we assume $\text{atoms}(t) \subseteq \text{dom}\,\sigma$, otherwise σ is for the duration of the computation extended to an encoding σ' which has this property

c' which behaves like c but whose type is derived from that of c by replacing δ by $unit \rightarrow \delta$.

This maybe not proves, but lets us very strongly believe:

The HCL cokernel is secure. ◂ Theorem 2.4

2.6.4 Mixing Modus Ponens, Instantiation, and Computation

Using the HCL cokernel comes with an overhead. Isabelle terms have to be converted into abstract machine terms for the purpose of computation, and after the computation has been performed, the resulting abstract machine terms have to be converted back to Isabelle terms. If the terms involved are big, the cost of converting between the Isabelle kernel and the cokernel universes can dominate the actual computing costs. In this subsection we scetch a simple feature of the HCL cokernel which drastically reduces traffic between kernel and cokernel and dramatically boosts performance in such a situation. The idea is that the cokernel assumes some reasoning responsibilities beyond computing equations.

Imagine an Isabelle theorem of the form:

$$\begin{array}{cccc} & f_1 & \equiv & g_1 \\ \Longrightarrow & f_2 & \equiv & g_2 \\ \Longrightarrow & f_3 & \equiv & g_3 \\ \vdots & \vdots & \vdots & \vdots \\ \Longrightarrow & f_n & \equiv & g_n \\ \Longrightarrow & \mathrm{p} & & \end{array}$$

One can understand such a theorem as an instruction book: First prove $f_1 \equiv g_1$. Then prove $f_2 \equiv g_2$. After that prove $f_3 \equiv g_3$. When you have finally proved $f_n \equiv g_n$, you have proved the theorem p.

For the cokernel, proving means computing. So the instruction to prove $f_1 \equiv g_1$ tells him to compute f_1 and g_1, and to get rid of the assumption $f_1 \equiv g_1$ if both are equal. This brings together the logical rule of modus ponens with computation. If we furthermore throw in variable instantiation, the mix becomes really powerful. Typically the theorem will contain several variables. Some of these variables are used as template parameters; instantiating them means adapting the theorem to the situation at hand. After these template parameters have been instantiated, the values of the other variables will often be uniquely determined by the requirement that all assumptions $f_i \equiv g_i$ must be true.

Upon choosing an i, the cokernel will take the following actions:

1. Compute f_i to f_i', and compute g_i to g_i'.

2. Try then to match g_i' with f_i' by instantiating the free variables in g_i' with terms which do not contain any of these free variables.

3. If this succeeds, remove the assumption $f_i \equiv g_i$ from the theorem and apply the found substitution throughout the whole theorem.

These actions are of course not executed on the Isabelle theorem, but on some theorem representation internal to the cokernel. The cokernel offers five securely accessible operations for this internal theorem representation:

1. Creation of an internal theorem from an Isabelle theorem.
2. Instantiation of variables of the internal theorem.
3. Elimination of assumptions of the internal theorem as described above.
4. Elimination of an assumption of the internal theorem by performing modus ponens with an Isabelle theorem. The assumptions of the Isabelle theorem are added as assumptions to the internal theorem.
5. Export of an internal theorem as an Isabelle theorem. This operation includes computing the proposition of the theorem as a whole before converting it into an Isabelle theorem.

The internal theorem is not represented by Isabelle terms, but is stored in the form of abstract machine terms. The substitutions involved in operation 2, 3 and 4 are not executed directly, but delayed. This works very much like the closure mechanism of the Barras machine. Because the elements of the image of the substitutions are all closed abstract machine terms that have already been computed, they can be tagged with the *Computed* constructor when substituted into a term that is about to be evaluated.

2.6.5 Polymorphic Linking

Currently the HCL is not very user-friendly. The user has to gather all the theorems needed for computing himself. This situation is worsened considerably by the fact that the HCL cokernel performs no instantiations of type variables. For example, look at the *If* constant. For computing with it the two equations in (2.4) must be passed to the cokernel. But they have to be instantiated to exactly the type of the *If* constant they are supposed to work on. Now the *If* constant is used with quite a lot of different types, and it is unacceptable that the user has to provide all needed instantiations of equations (2.4).

Polymorphic linking solves this problem. The polymorphic linker is a wrapper around the cokernel. It allows to create a *polymorphic computer* which internally manages an ordinary computer. The polymorphic computer is created from a list of polymorphic theorems. When it is asked to compute a term it checks if new instances of these polymorphic theorems are needed and if so, updates the internal computer with those new theorems. The polymorphic computer provides the same interface as an ordinary computer. It also supports the feature presented in the previous subsection of mixing logical reasoning and computing.

CHAPTER **3**

Proving Bounds for Real Linear Programs

The sky's the limit if you have a roof over your head.
— Sol Hurok

Contents

3.1 Overview

In the next chapter our work will culminate in proving that about 92.5% of what we call *the basic linear programs* are infeasible. A prerequisite for this result is the method of bounding the objective function of a linear program (LP) that we present in this chapter.

The method is due to Hales [12] where he describes how to obtain an arbitrarily precise upper bound for the maximum value of the objective function of an LP. Our contribution is to transfer this method into the rigid context of mechanical theorem provers, specifically our favorite one, Isabelle/HOL [25]. The burden of calculating the upper bound is delegated to an LP solver that needs not to be trusted. Instead, the LP solver delivers a small certificate to Isabelle/HOL that can be checked cheaply. Furthermore, there is no need to delve into the details of the actual method of optimizing an LP, which is usually the Simplex method. These details just do not matter for the theorem prover.

We first describe the basic idea of the method. Then we introduce the notion of *finite matrices* and explain why these are our representation of choice for linear programs. Finite matrices can be fitted into the system of numeric axiomatic type classes in Isabelle/HOL via the algebraic concept of *lattice-ordered rings*. Checking the certificate from the external LP solver is basically a calculation involving finite matrices. The matrices we have to deal with are sparse, therefore we introduce a sparse matrix representation of finite matrices.

We have presented most of the material in this chapter already in [26]. New additions are Sections 3.5, 3.7 and 3.8.

3.2 The Basic Idea

There are quite a lot of different ways to state a linear programming problem [32, sect. 7.4], which are all general in the sense that *every* linear programming problem can be stated that way. Here is one such way: a linear program consists of a matrix $A \in \mathbb{R}^{m\times n}$, a row vector $c \in \mathbb{R}^{1\times n}$ and a column vector $b \in \mathbb{R}^{m\times 1}$. The goal is to maximize the objective function

$$x \longmapsto cx, \quad x \text{ feasible}, \tag{3.1}$$

where x is called *feasible* iff $x \in \mathbb{R}^{n\times 1}$ and $Ax \le b$ holds. Note that we are dealing with matrix inequality here: $X \le Y$ for two matrices X and Y iff every matrix element of X is less than or equal to the corresponding element of Y.

Usually, the above stated goal really encompasses several goals / questions:

1. Find out if there exists any feasible x at all (otherwise the LP is called *infeasible*).
2. Find out if there is a feasible $x_{\max}$ such that $cx_{\max} \ge cx$ for any feasible x, and calculate this $x_{\max}$.
3. Calculate $M = \sup\{cx \mid x \text{ is feasible}\}$.

Note that $M = -\infty$ iff the answer to the first question is no. And $M = \infty$ iff the answer to the first question is yes and the answer to the second question is no. If $M < \infty$ then the LP is called *bounded*. Linear programming software is good at answering all those questions and at exhibiting (approximately) such an $x_{\max}$ if it exists. Our goal is more modest in some ways, but more demanding in others: *assuming a priori bounds for the feasible region, that is assuming $l \le x \le u$ for all feasible x with a priori known bounds l and u, actually prove within Isabelle/HOL that $M \le K$, where we can choose K arbitrarily close to M.* In particular, we do not want to calculate $x_{\max}$, but just want to approximate M as precise as we wish for. Furthermore, we can assume $M \neq \infty$ because of

$$M \leq \sum_{i=1}^{n} |c_{1i}| \max\{|l_{i1}|, |u_{i1}|\} < \infty \ . \tag{3.2}$$

It might seem that our goal can be accomplished trivially by setting K to the above sum. But of course this is not the case, as K is probably not a particularly good approximation for M, and there is nothing in the above inequality telling us how to get a better approximation in case we need one.

3.2.1 Reducing the case $M = -\infty$ to the case $-\infty < M < \infty$

The case of an infeasible LP can be reduced to the case of a feasible LP [12]. We will give a more detailed description here than the one found in [12].

Remember that we are only considering LPs for which we know l and u s.t.

$$Ax \leq b \Longrightarrow l \leq x \leq u \ . \tag{3.3}$$

In this subsection we additionally require A to fulfill the inequality

$$Ax \leq 0 \Longrightarrow x = 0 \ . \tag{3.4}$$

This can easily be arranged by replacing A and b by $\tilde{A}$ and $\tilde{b}$ where

$$\tilde{A} = \begin{pmatrix} A \\ I_n \\ -I_n \end{pmatrix} \quad \text{and} \quad \tilde{b} = \begin{pmatrix} b \\ u \\ -l \end{pmatrix} \ . \tag{3.5}$$

$I_n \in \mathbb{R}^{n \times n}$ denotes the identity matrix.

Now let us assume that for the given LP both (3.3) and (3.4) hold. We can construct for any $K \in \mathbb{R}$ a modified LP with objective function

$$x' = \begin{pmatrix} x \\ t \end{pmatrix} \longmapsto cx + Kt, \quad x' \text{ feasible}, \tag{3.6}$$

where $x' \in \mathbb{R}^{n+1}$ is called feasible with respect to the modified LP iff

$$Ax + tb \leq b \quad \text{and} \quad 0 \leq t \leq 1 \ . \tag{3.7}$$

◀ Theorem 3.1

$$\begin{pmatrix} x \\ 1 \end{pmatrix} \textit{ is feasible} \Longleftrightarrow x = 0 \ , \tag{3.8}$$

$$0 \leq t < 1 \Longrightarrow \left(\begin{pmatrix} x \\ t \end{pmatrix} \textit{ is feasible} \Longleftrightarrow x/(1-t) \textit{ is feasible} \right) \ . \tag{3.9}$$

On the left hand side of above equivalences we talk about feasibility with respect to the modified LP, on the right hand side about feasibility with respect to the original LP.

Proof. To show (3.8) in the direction from left to right one needs the fact that A fulfills (3.4). The rest ist obvious by just expanding the respective definition of feasibility. □

Theorem 3.2 ► *Defining* $M' := \sup\{cx + Kt \mid x' = \begin{pmatrix} x \\ t \end{pmatrix},\ x' \text{ feasible}\}$ *yields*

$$-\infty < \max\{M,K\} = M' < \infty\ . \tag{3.10}$$

As a special case follows

$$M = -\infty \Longrightarrow M' = K\ . \tag{3.11}$$

Proof. Because of (3.8) we have $M' \geq K$, in particular $M' > -\infty$. Considering $t = 0$ in (3.9) gives us $M' \geq M$. From (3.9) and (3.3) in the case $t \neq 1$ and (3.8) in the case $t = 1$ we obtain bounds for x':

$$x' = \begin{pmatrix} x \\ t \end{pmatrix} \text{ is feasible } \Longrightarrow l^- \leq (1-t)l \leq x \leq (1-t)u \leq u^+\ .$$

Here l^- denotes the *negative part* of l which results from l by replacing every positive matrix element by 0. Similarly, the *positive part* u^+ results from u by replacing every negative element by 0. We conclude $M' < \infty$.

So far we have shown $-\infty < \max\{M,K\} \leq M' < \infty$. To complete the proof, we need to show $\max\{M,K\} \geq M'$. We will proceed by case distinction.

Assume $M \geq K$. We show that for any feasible $x' = \begin{pmatrix} x \\ t \end{pmatrix}$, $M \geq cx + Kt$, and therefore $M \geq M'$. This is obvious in the case $t = 1$, the feasibility of x' accompanied by the equivalence (3.8) forces x to be zero. In the case $t \neq 1$, (3.9) implies that $x/(1-t)$ is feasible with respect to the original LP. But this is just what we claim:

$$M \geq c(x/(1-t)) \Longrightarrow M \geq cx + tM \geq cx + tK\ .$$

Now assume $M < K$. Assume further $M' > K$. Because of $-\infty < M' < \infty$ there is a feasible $x' = \begin{pmatrix} x \\ t \end{pmatrix}$ s.t. $M' = cx + Kt$. For $t = 1$ we would have again $x = 0$ and therefore the contradiction $K < M' = K$. Finally $0 \leq t < 1$ also leads to a contradiction:

$$M' = cx + Kt \leq cx + M't \Longrightarrow M' \leq c(x/(1-t)) \leq M < K \leq M'\ .$$

Therefore the only possibility is $M' \leq K$. □

From now on we will assume that we are dealing with feasible, bounded LPs, that is with LPs for which we know $-\infty < M < \infty$.

3.2.2 The case $-\infty < M < \infty$

This case is the heart of the method. Again we construct a modified LP. The original LP is called the *primal* LP, the modified LP is called the *dual* LP. The objective function of the dual LP

$$y \longmapsto yb, \quad y \text{ feasible},$$

is to be minimized. Here $y \in \mathbb{R}^{1\times m}$ is called feasible iff $yA = c$ and $y \geq 0$ holds.

Theorem 3.3 ► *Any feasible y induces an upper bound on M:*

$$yb \geq M\ . \tag{3.12}$$

Proof. For any feasible x we have

$$y\,b \geq y\,(A\,x) = (y\,A)\,x = c\,x \ . \tag{3.13}$$

□

But is there such a feasible y so that we can utilize (3.12)? And if there is, can we accomplish $y\,b = M$ by carefully choosing y? The well-known answer to both questions is yes:

◂ Theorem 3.4

Define $M' := \inf\{y\,b \,|\, y\,A = c$ and $y \geq 0\}$. Then

$$-\infty < M = M' < \infty \ . \tag{3.14}$$

Furthermore, choose a feasible y such that $M' = y\,b$. Then

$$\text{card}\,\{i \in \mathbb{N} \,|\, 1 \leq i \leq m \text{ and } y_{1i} > 0\} \leq n \ . \tag{3.15}$$

Proof. Corollary 7.1g and 7.1l in [32]. □

Now the basic idea of our method can be described as follows. First, form the dual LP. Then use an external LP solver to solve the dual LP for an optimal y. This optimal y serves as a *certificate*. In our application, where typically $m \approx 2000$ and $n \approx 200$, y will be *sparse*, as inequality (3.15) tells us. Finally, use (3.12) to verify our desired upper bound $M \leq K = y\,b$.

This basic idea is complicated by the fact that we are dealing with *real* data and *numerical* algorithms. The external LP solver does not return an y such that $y\,A = c$ and $y \geq 0$, but rather an y such that $y\,A \approx c$ and $y \gtrapprox 0$. In order to obtain a provably upper bound on M, one has to take (3.3) into account. Furthermore the input data A, b and c need not to be given as exact numerical data either, for example an element of A could equal π. The way to deal with these complications is to use *interval arithmetic*, not only for real numbers, but also for real matrices.

3.3 Finite Matrices

Anyone who wants to implement the method outlined in the previous section faces the problem of how to represent linear programs. This problem is prominent outside of the realm of mechanical theorem proving, too: designers of linear programming packages typically provide various ways of input of data to the LP algorithms these packages provide, one can normally choose at least between dense and sparse representations of the data. The issue is to provide a certain convenience of dealing with the data without compromising the efficiency of the LP algorithms by too much overhead.

Our situation is different: we need to reason within our mechanical theorem proving environment Isabelle/HOL why our computations lead to a correct result, therefore we need a good representation of LPs for reasoning about them. Of course we also need to compute efficiently. But we should avoid mixing up those two issues if we can. The reasoning in the previous section has used matrices and the properties of matrix operations like associativity of matrix multiplication extensively. Therefore representing LPs within Isabelle/HOL as matrices is a good idea.

So how exactly does one represent matrices in higher-order logic? Obviously, matrices should be a type, but how does one deal with the dimension of a matrix? HOL does not have *dependent types*, so it seems impossible to have a parametrized family of types where the dimension of the matrix would be the parameter. But it is: one possibility that is pursued by John Harrison in the 2005 version of his Hol-light system is to represent the needed parameter by type variables! He uses this representation in order to formalize multivariate calculus. But in our case this idea cannot be used without causing serious problems later when we turn our attention to sparse matrices.

Another possibility is to represent the dimension of a matrix by a predicate that carves the set of all matrices of this dimension out of a certain bigger, already existing type. This is a common technique to overcome the absence of dependent types in HOL [18]. This approach could work like this:[1]

type $\alpha\ M = nat \times nat \times (nat \Rightarrow nat \Rightarrow \alpha)$

constdef
$Mequiv :: (\alpha\ M * \alpha\ M)\ set$
$Mequiv \equiv \{((m,n,f),(m,n,g)) \mid \forall\, j\, i.\, (j < m \wedge i < n) \longrightarrow f\, j\, i = g\, j\, i\}$

typedef $\alpha\ matrix = UNIV\ //\ Mequiv$ (3.16)

constdef
$is\text{-}matrix :: nat \Rightarrow nat \Rightarrow \alpha\ matrix \Rightarrow bool$
$is\text{-}matrix\ m\ n\ A \equiv \exists f.\, (m,n,f) \in Rep\text{-}matrix\, A.$

In (3.16) $\alpha\ matrix$ is the bigger type, and $is\text{-}matrix\, m\, n$ acts as the predicate that carves out all matrices consisting of m rows and n columns. Here matrices are modelled as equivalence classes [31] of triples (m,n,f) where m denotes the number of rows, n the number of columns and f a function from indices to matrix elements. The set of these equivalence classes is denoted by $UNIV\ //\ Mequiv$. With this formalization of matrices an error element comes for free: there is exactly one matrix $Error$ such that

$$is\text{-}matrix\, 0\, 0\, Error \tag{3.17}$$

holds. When adding matrices A and B which fulfill

$$\exists\, m\, n\, .\ (is\text{-}matrix\ m\ n\ A) \wedge (\neg\ is\text{-}matrix\ m\ n\ B) \tag{3.18}$$

and when multiplying matrices A and B for which

$$\exists\, m\, n\, u\, v.\ (is\text{-}matrix\ m\ n\ A) \wedge (is\text{-}matrix\ u\ v\ B) \wedge (n \neq u) \tag{3.19}$$

holds, the matrix $Error$ is returned to signal that the operands do not belong to the natural domain of addition and multiplication, respectively.

Still, this approach is not entirely satisfying: in Isabelle/HOL there exists a large number of theorems that are valid for types that form a group or a ring. The fact that a type can be viewed as such an algebraic structure is formulated via the concept of

[1] Here and in the following we deviate slightly from actual Isabelle/HOL syntax for various reasons, the most important being formatting; the actual Isabelle/HOL user will have no difficulty translating the given theory snippets to proper Isabelle/HOL syntax.

axiomatic type classes [30]. But α *matrix* in (3.16) with the suggested error signaling definition of addition does not even form a group, because there is no matrix *Zero* with

$$\forall A.\, A + Zero = A \ , \tag{3.20}$$

but rather a whole family $Zero\, m\, n$ such that

$$\forall A.\, \textit{is-matrix}\, m\, n\, A \longrightarrow A + (Zero\, m\, n) = A \ . \tag{3.21}$$

Therefore we advocate a different approach that exploits the fact that the matrix elements commonly used in mathematics [20] themselves carry an algebraic structure, that of a ring, which always contains a zero. We define α *matrix* to be the type formed by all infinite matrices that have only finitely many non-zero elements of type α:

$$\begin{aligned} &\textbf{type}\ \alpha\ \textit{infmatrix} = \textit{nat} \Rightarrow \textit{nat} \Rightarrow \alpha \\ &\textbf{typedef}\ \alpha\ \textit{matrix} = \{f :: (\alpha :: \textit{zero})\textit{infmatrix} \mid \textit{finite}\{(j,i) \mid f\, j\, i \neq 0\}\} \ . \end{aligned} \tag{3.22}$$

Hence we choose the name *finite matrices* for objects of type α *matrix*. Note the restriction $\alpha :: \textit{zero}$ in (3.22). This means that the elements of a matrix cannot have just any type but only a type that is an instance of the axiomatic type class *zero* and has thus an element denoted by 0. Of course this is not a real restriction on the type; any type can be declared to be an instance of the axiomatic type class *zero*.

3.3.1 Dimension of a Finite Matrix

The dimension of a finite matrix deviates from the notion of dimension that one is used to. Because we did *not* encode the number of rows and columns explicitly in the representation of a finite matrix as we did in (3.16), we have to recover the dimension of a finite matrix by extensionality:

$$\begin{aligned} &\textbf{constdefs} \\ &\quad \textit{nrows} :: \alpha\ \textit{matrix} \Rightarrow \textit{nat} \\ &\quad \textit{nrows}\, A \equiv \textit{LEAST}\, m.\, \forall\, j\, i.\, m \leq j \longrightarrow (\textit{Rep-matrix}\, A\, j\, i = 0) \\ &\quad \textit{ncols} :: \alpha\ \textit{matrix} \Rightarrow \textit{nat} \\ &\quad \textit{ncols}\, A \equiv \textit{LEAST}\, n.\, \forall\, j\, i.\, n \leq i \longrightarrow (\textit{Rep-matrix}\, A\, j\, i = 0) \\ &\quad \textit{is-matrix} :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \alpha\ \textit{matrix} \Rightarrow \textit{bool} \\ &\quad \textit{is-matrix}\ m\ n\ A \equiv \textit{nrows}\, A \leq m \wedge \textit{ncols}\, A \leq n \ . \end{aligned} \tag{3.23}$$

The expression $\textit{LEAST}\, x.\, P\, x$ equals the least x such that $P\, x$ holds. The definition of the type α *matrix* has introduced two automatically defined functions *Rep-matrix* and *Abs-matrix*

$$\begin{aligned} &\textbf{consts} \\ &\quad \textit{Rep-matrix} :: \alpha\ \textit{matrix} \Rightarrow \alpha\ \textit{infmatrix} \\ &\quad \textit{Abs-matrix} :: \alpha\ \textit{infmatrix} \Rightarrow \alpha\ \textit{matrix} \end{aligned} \tag{3.24}$$

that convert between finite matrices and infinite matrices. They enjoy the following crucial properties:

$$(A = B) = (\forall\, j\, i.\, \textit{Rep-matrix}\, A\, j\, i = \textit{Rep-matrix}\, B\, j\, i) \ , \tag{3.25}$$

$$\exists_1 f.\, A = \textit{Abs-matrix}\, f \ , \tag{3.26}$$

$$\textit{Abs-matrix}\,(\textit{Rep-matrix}\,A) = A \ , \tag{3.27}$$

$$\textit{finite}\,\{(j,i) \mid \textit{Rep-matrix}\,A\,j\,i \neq 0\} \ , \tag{3.28}$$

$$\textit{finite}\,\{(j,i) \mid f\,j\,i \neq 0\} \Longrightarrow \textit{Rep-matrix}\,(\textit{Abs-matrix}\,f) = f \ . \tag{3.29}$$

Thus $\textit{Rep-matrix}\,A\,j\,i$ denotes the matrix element of A in row j and column i. Note that the first row is row 0, likewise for columns.

Let us return to the definitions in (3.23). The definition of *is-matrix* implies that a matrix has not exactly one dimension, but infinitely many! Therefore there is no need for signaling an error due to incompatibility of dimensions: for any two matrices A and B one shows

$$\exists m.\ \textit{is-matrix}\,m\,m\,A \wedge \textit{is-matrix}\,m\,m\,B \ . \tag{3.30}$$

The intuition behind (3.30) is that every matrix can be viewed as a square matrix of dimension m as long as m is large enough: one just needs to fill up the missing rows and columns with zeros.

The need for an *Error* matrix has vanished, but one can still use (3.17) to uniquely define a matrix. This time, we denote that matrix by 0:

$$\forall A.\ (A = 0) = (\textit{is-matrix}\,0\,0\,A) \ . \tag{3.31}$$

Another possibility of defining 0 is given by the following theorem:

$$\forall A.\ (A = 0) = (\forall m\,n.\ \textit{is-matrix}\,m\,n\,A) \ . \tag{3.32}$$

We will see that 0 is actually the proper name for this matrix.

3.3.2 Lifting Unary Operators

In this subsection we look at how to define an unary operator U on matrices,

$$U :: \alpha\ \textit{matrix} \Rightarrow \beta\ \textit{matrix} \ , \tag{3.33}$$

by lifting an unary operator u on matrix elements,

$$u :: \alpha \Rightarrow \beta \ . \tag{3.34}$$

The first step is to lift u to infinite matrices:

constdef

$$\begin{aligned} &\textit{apply-infmatrix} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha\ \textit{infmatrix} \Rightarrow \beta\ \textit{infmatrix}) \\ &\textit{apply-infmatrix}\ u \equiv \lambda\, f\,j\,i.\ u\,(f\,j\,i) \ , \end{aligned} \tag{3.35}$$

which results in the lifting property

$$(\textit{apply-infmatrix}\,u\,f)\,j\,i = u\,(f\,j\,i) \ . \tag{3.36}$$

Its proof is apparent from the definition of *apply-infmatrix*.

Now the unary lifting operator *apply-matrix* can be defined by first lifting u to infinite matrices, and then to finite matrices:

constdef

$$\begin{aligned} &\textit{apply-matrix} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha\ \textit{matrix} \Rightarrow \beta\ \textit{matrix}) \\ &\textit{apply-matrix}\ u \equiv \lambda A.\ \textit{Abs-matrix}\,(\textit{apply-infmatrix}\,u\,(\textit{Rep-matrix}\ A)) \ . \end{aligned} \tag{3.37}$$

What is the lifting property for *apply-matrix*? A first guess yields

$$\textit{Rep-matrix}(\textit{apply-matrix}\, u\, A)\, j\, i = u\,(\textit{Rep-matrix}\, A\, j\, i) \ . \tag{3.38}$$

But this is false (in the sense that we cannot prove it in HOL)! To see why, consider $\alpha = \beta = \mathit{int}$ and $u = \lambda\, x.\, 1$. Then we have

$$\textit{apply-infmatrix}\, u\,(\textit{Rep-matrix}\, A) = \begin{pmatrix} 1 & 1 & \cdots \\ 1 & 1 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \neq \textit{Rep-matrix}\, B \tag{3.39}$$

for all matrices A and any matrix B. But there is a simple condition on u that turns out to be sufficient and necessary to prove (3.38):

$$u\, 0 = 0 \Longrightarrow \textit{Rep-matrix}(\textit{apply-matrix}\, u\, A)\, j\, i = u\,(\textit{Rep-matrix}\, A\, j\, i) \ . \tag{3.40}$$

This is easily provable using (3.37), (3.28), (3.29) and (3.36).

3.3.3 Lifting Binary Operators

Just as we have defined a unary lifting operator *apply-matrix*, we can define similarly a binary lifting operator *combine-matrix*:

$$\begin{array}{l} \textbf{constdef} \\ \quad \textit{combine-infmatrix} :: \\ \qquad (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\alpha\ \textit{infmatrix} \Rightarrow \beta\ \textit{infmatrix} \Rightarrow \gamma\ \textit{infmatrix}) \\ \quad \textit{combine-infmatrix}\ v \equiv \lambda\, f\, g\, j\, i.\, v\,(f\, j\, i)\,(g\, j\, i) \ , \end{array} \tag{3.41}$$

$$\begin{array}{l} \textbf{constdef} \\ \quad \textit{combine-matrix} :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\alpha\ \textit{matrix} \Rightarrow \beta\ \textit{matrix} \Rightarrow \gamma\ \textit{matrix}) \\ \quad \textit{combine-matrix}\ v \equiv \\ \qquad \lambda\, A\, B.\, \textit{Abs-matrix}(\textit{combine-infmatrix}\, v\,(\textit{Rep-matrix}\, A)\,(\textit{Rep-matrix}\, B)) \ , \end{array} \tag{3.42}$$

The lifting property for *combine-matrix* reads

$$\begin{array}{l} v\, 0\, 0 = 0 \Longrightarrow \textit{Rep-matrix}(\textit{combine-matrix}\, v\, A\, B)\, j\, i = \\ \qquad\qquad v\,(\textit{Rep-matrix}\, A\, j\, i)\,(\textit{Rep-matrix}\, B\, j\, i) \ . \end{array} \tag{3.43}$$

Lifting binary operators passes on commutativity and associativity. Defining

$$\begin{array}{l} \textbf{constdefs} \\ \quad \textit{commutative} :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \textit{bool} \\ \quad \textit{commutative}\ v \equiv \forall\, x\, y.\, v\, x\, y = v\, y\, x \\ \quad \textit{associative} :: (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \textit{bool} \\ \quad \textit{associative}\ v \equiv \forall\, x\, y\, z.\, v\,(v\, x\, y)\, z = v\, x\,(v\, y\, z) \end{array} \tag{3.44}$$

we can formulate this propagation concisely:

$$\begin{array}{l} \textit{commutative}\, v \Longrightarrow \textit{commutative}(\textit{combine-matrix}\, v) \ , \\ [\![v\, 0\, 0 = 0;\ \textit{associative}\, v]\!] \Longrightarrow \textit{associative}(\textit{combine-matrix}\, v) \ . \end{array} \tag{3.45}$$

You might be surprised that the propagation of commutativity does not require $v\, 0\, 0 = 0$, which is due to the idiosyncrasies of the definite description operator that is hidden in *Abs-matrix*.

3.3.4 Matrix Multiplication

We need one last lifting operation, the most interesting one: given two binary operators addition and multiplication on the matrix elements, define the matrix product induced by those two operators. As a basic tool we first define by primitive recursion a fold operator that acts on sequences:

$$\begin{array}{l}\textbf{const}\ \mathit{foldseq} :: (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow (\mathit{nat} \Rightarrow \alpha) \Rightarrow \mathit{nat} \Rightarrow \alpha \\ \textbf{primrec} \\ \quad \mathit{foldseq}\ f\ s\ 0 = s\, 0 \\ \quad \mathit{foldseq}\ f\ s\ (\mathit{Suc}\, n) = f\,(s\,0)\,(\mathit{foldseq}\, f\,(\lambda k.\, s\,(\mathit{Suc}\, k))\, n)\end{array} \tag{3.46}$$

For illustration purposes, assume $s = (s_1, s_2, s_3, s_4, \ldots, s_n, 0, 0, 0, \ldots)$. Then

$$\begin{array}{l}\mathit{foldseq}\ f\ s\ 0 = s_1\ , \\ \mathit{foldseq}\ f\ s\ 1 = f\, s_1\, s_2\ , \\ \mathit{foldseq}\ f\ s\ 2 = f\, s_1\,(f\, s_2\, s_3)\ , \\ \mathit{foldseq}\ f\ s\ 3 = f\, s_1\,(f\, s_2\,(f\, s_3\, s_4))\ , \\ \mathit{foldseq}\ f\ s\ n = f\, s_1\,(f\, s_2\,(\ldots (f\, s_n\, 0)\ldots))\ , \\ \mathit{foldseq}\ f\ s\ (n+1) = f\, s_1\,(f\, s_2\,(\ldots (f\, s_n\,(f\,0\,0)\ldots))\ \text{and so on.}\end{array} \tag{3.47}$$

Note that if $f\,0\,0 = 0$ the above sequence converges:

$$f\,0\,0 = 0 \Longrightarrow \forall m.\, n \le m \longrightarrow \mathit{foldseq}\, f\, s\, m = \mathit{foldseq}\, f\, s\, n\ . \tag{3.48}$$

Now we are prepared to deal with matrix multiplication:

$$\begin{array}{l}\textbf{constdef} \\ \quad \mathit{mult\text{-}matrix\text{-}n} :: \mathit{nat} \Rightarrow (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\gamma \Rightarrow \gamma \Rightarrow \gamma) \Rightarrow \\ \qquad\qquad \alpha\ \mathit{matrix} \Rightarrow \beta\ \mathit{matrix} \Rightarrow \gamma\ \mathit{matrix} \\ \quad \mathit{mult\text{-}matrix\text{-}n}\ n\ \mathit{mult}\ \mathit{add}\ A\ B \equiv \mathit{Abs\text{-}matrix}\,(\lambda\, j\, i. \\ \qquad \mathit{foldseq}\, \mathit{add}\,(\lambda k.\, \mathit{mult}(\mathit{Rep\text{-}matrix}\, A\, j\, k)(\mathit{Rep\text{-}matrix}\, B\, k\, i))\, n)\end{array} \tag{3.49}$$

The idea of $\mathit{mult\text{-}matrix\text{-}n}\, n\, \mathit{mult}\, \mathit{add}\, A\, B$ is to consider only the first n columns of A and the first n rows of B when calculating the matrix product. Of course the matrix product should be independent of n. We achieve this by setting

$$\mathit{mult\text{-}matrix}\, \mathit{mult}\, \mathit{add} \equiv \lim_{n\to\infty} \mathit{mult\text{-}matrix\text{-}n}\, n\, \mathit{mult}\, \mathit{add}\ , \tag{3.50}$$

which is due to (3.48) well-defined if $\forall x.\, \mathit{mult}\, x\, 0 = \mathit{mult}\, 0\, x = \mathit{add}\, 0\, 0 = 0$ holds:

$$\begin{array}{l}\textbf{constdef} \\ \quad \mathit{mult\text{-}matrix} :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\gamma \Rightarrow \gamma \Rightarrow \gamma) \Rightarrow \\ \qquad\qquad \alpha\ \mathit{matrix} \Rightarrow \beta\ \mathit{matrix} \Rightarrow \gamma\ \mathit{matrix} \\ \quad \mathit{mult\text{-}matrix}\ \mathit{mult}\ \mathit{add}\ A\ B \equiv \\ \qquad \mathit{mult\text{-}matrix\text{-}n}(\max\,(\mathit{ncols}\, A)\,(\mathit{nrows}\, B))\, \mathit{mult}\, \mathit{add}\, A\, B\ .\end{array} \tag{3.51}$$

Again, we have a lifting property:

$$\begin{array}{l}[\![\forall x.\, \mathit{mult}\, x\, 0 = 0 \wedge \mathit{mult}\, 0\, x = 0;\ \mathit{add}\, 0\, 0 = 0]\!] \Longrightarrow \\ \quad \mathit{Rep\text{-}matrix}(\mathit{mult\text{-}matrix}\, \mathit{mult}\, \mathit{add}\, A\, B)\, j\, i = \mathit{foldseq}\, \mathit{add} \\ \qquad (\lambda k.\, \mathit{mult}(\mathit{Rep\text{-}matrix}\, A\, j\, k)(\mathit{Rep\text{-}matrix}\, B\, k\, i))(\mathit{max}(\mathit{ncols}\, A)(\mathit{nrows}\, B))\,.\end{array} \tag{3.52}$$

Finally, let us examine what properties of element addition and element multiplication induce distributivity and associativity of *mult-matrix*.

3.3.4.1 Distributivity

We distinguish between left and right distributivity:[2]

constdefs

$$\begin{array}{l} \textit{r-distributive} :: (\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \beta \Rightarrow \beta) \Rightarrow \textit{bool} \\ \textit{r-distributive mult add} \equiv \forall a\, u\, v.\, \textit{mult}\, a\, (\textit{add}\, u\, v) = \textit{add}\, (\textit{mult}\, a\, u)\, (\textit{mult}\, a\, v) \quad \textit{l-} \\ \textit{distributive} :: (\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \textit{bool} \\ \textit{l-distributive mult add} \equiv \forall a\, u\, v.\, \textit{mult}\, (\textit{add}\, u\, v)\, a = \textit{add}\, (\textit{mult}\, u\, a)\, (\textit{mult}\, v\, a) \end{array} \tag{3.53}$$

Distributivity of *mult* over *add* lifts to distributivity of *mult-matrix mult add* over *combine-matrix add* if *add* is associative and commutative and both *add* and *mult* behave as expected with respect to 0:

$$\begin{array}{l} [\![\, \textit{l-distributive mult add; associative add; commutative add;} \\ \quad \forall x.\, \textit{mult}\, x\, 0 = 0 \wedge \textit{mult}\, 0\, x = 0;\ \textit{add}\, 0\, 0 = 0 \,]\!] \\ \Longrightarrow \textit{l-distributive}\ (\textit{mult-matrix mult add})\ (\textit{combine-matrix add}) \ , \\ [\![\, \textit{r-distributive mult add; associative add; commutative add;} \\ \quad \forall x.\, \textit{mult}\, x\, 0 = 0 \wedge \textit{mult}\, 0\, x = 0;\ \textit{add}\, 0\, 0 = 0 \,]\!] \\ \Longrightarrow \textit{r-distributive}\ (\textit{mult-matrix mult add})\ (\textit{combine-matrix add}) \ . \end{array} \tag{3.54}$$

3.3.4.2 Associativity

We state the law of associativity for *mult-matrix* in a very general form:

$$\begin{array}{l} [\![\, \forall a.\, \textit{mult}_1\, a\, 0 = 0;\ \forall a.\, \textit{mult}_1\, 0\, a = 0;\ \forall a.\, \textit{mult}_2\, a\, 0 = 0;\ \forall a.\, \textit{mult}_2\, 0\, a = 0; \\ \quad \textit{add}_1\, 0\, 0 = 0;\ \textit{add}_2\, 0\, 0 = 0; \\ \quad \forall a\, b\, c\, d.\, \textit{add}_2\, (\textit{add}_1\, a\, b)\, (\textit{add}_1\, c\, d) = \textit{add}_1\, (\textit{add}_2\, a\, c)\, (\textit{add}_2\, b\, d); \\ \quad \forall a\, b\, c.\, \textit{mult}_2\, (\textit{mult}_1\, a\, b)\, c = \textit{mult}_1\, a\, (\textit{mult}_2\, b\, c); \\ \quad \textit{associative add}_1;\ \textit{associative add}_2; \\ \quad \textit{l-distributive mult}_2\ \textit{add}_1;\ \textit{r-distributive mult}_1\ \textit{add}_2 \,]\!] \\ \Longrightarrow\ \textit{mult-matrix mult}_2\ \textit{add}_2\ (\textit{mult-matrix mult}_1\ \textit{add}_1\ A\ B)\ C = \\ \qquad \textit{mult-matrix mult}_1\ \textit{add}_1\ A\ (\textit{mult-matrix mult}_2\ \textit{add}_2\ B\ C) \ . \end{array} \tag{3.55}$$

For $\textit{mult} = \textit{mult}_1 = \textit{mult}_2$ and $\textit{add} = \textit{add}_1 = \textit{add}_2$ this simplifies to

$$\begin{array}{l} [\![\, \forall a.\, \textit{mult}\, a\, 0 = 0;\ \forall a.\, \textit{mult}\, 0\, a = 0;\ \textit{add}\, 0\, 0 = 0; \\ \quad \textit{associative add; commutative add; associative mult;} \\ \quad \textit{l-distributive mult add; r-distributive mult add} \,]\!] \\ \Longrightarrow\ \textit{associative}\ (\textit{mult-matrix mult add}) \ . \end{array} \tag{3.56}$$

3.3.5 Lattice-Ordered Rings

Paulson describes in [30] how numerical theories like the theory of integers or the theory of reals can be organized in Isabelle/HOL using axiomatic type classes. For example both integers and reals form a ring, therefore Paulson recommends to prove theorems that are implied purely by ring properties only once, and then to prove that both types *int* and the type *real* are an instance of the axiomatic type class *ring*.

Birkhoff points out [4, chapt. 17] that for a fixed n the ring of all $n \times n$ square matrices forms a latticed-ordered ring in a natural way. The same is true for our finite matrices! Therefore it suggests itself to establish an axiomatic type class *lordered-ring* that captures the property of a type to form a lattice-ordered ring. Of

[2] Our convention is that left distributivity means that the factor is distributed over the left sum, *not* that the left factor is the one that gets distributed.

course *lordered-ring* should be integrated with the other type classes like *ring* and *ordered-ring* of Isabelle/HOL to maximize theorem reuse. Two major changes along with minor modifications were necessary to the original hierarchy of type classes as described in [30]:

1. The original type class *ring* demanded both the existence of a multiplicative unit element and the commutativity of multiplication. But our finite matrices do not have such a multiplicative unit element, nor is multiplication of finite matrices a commutative operator. Nevertheless, finite matrices still form a ring in common mathematical terminology. Therefore the original type class *ring* was renamed to become *comm-ring-1* and new type classes *ring*, *ring-1* and *comm-ring* were introduced, suitable for rings that do not necessarily possess a 1 and/or are not commutative.

2. All ordered algebraic structures contained in the original hierarchy were linearly ordered. The natural (elementwise) order for finite matrices is a proper partial order, actually a lattice order. Therefore we enriched the hierarchy with type classes that model partially ordered algebraic systems like partially ordered groups and rings, or lattice-ordered groups and rings. For this we follow largely [7], [4].

A type α is an instance of the axiomatic type class *lordered-ring* iff

ring α is a ring with addition $+$, subtraction $-$, additive inverse $-$, multiplication $*$, zero 0,

lattice α is a lattice with partial order $\le$ and operators *join* and *meet*,

monotonicity addition and multiplication are monotone:

$$a \le b \longrightarrow c + a \le c + b \ , \tag{3.57}$$

$$a \le b \land 0 \le c \longrightarrow a * c \le b * c \land c * a \le c * b \ . \tag{3.58}$$

Both *int* and *real* are instances of *lordered-ring*:

$$\begin{array}{l}\textbf{instance } \mathit{int} :: \mathit{lordered\text{-}ring} \\ \textbf{instance } \mathit{real} :: \mathit{lordered\text{-}ring}\ .\end{array} \tag{3.59}$$

Our goal is to prove

$$\textbf{instance } \mathit{matrix} :: (\mathit{lordered\text{-}ring})\ \mathit{lordered\text{-}ring}\ . \tag{3.60}$$

The above meta theorem has the following meaning (which is not legal Isabelle syntax):

$$(\textbf{instance } \alpha :: \mathit{lordered\text{-}ring}) \Longrightarrow (\textbf{instance } \alpha\ \mathit{matrix} :: \mathit{lordered\text{-}ring})\ . \tag{3.61}$$

Of course, in order to prove (3.60), one first has to define 0, $+$, $*$ etc. for objects of type *matrix*. The zero matrix is easy to define:

$$\begin{array}{l}\textbf{instance } \mathit{matrix} :: (\mathit{zero})\ \mathit{zero} \\ \textbf{def (overloaded)} \\ \quad 0 \equiv \mathit{Abs\text{-}matrix}(\lambda\, j\, i.\, 0)\ .\end{array} \tag{3.62}$$

It is simple to show that this is actually the 0 we refer to in (3.31) and (3.32).

Addition +, multiplication $*$, subtraction $-$, unary minus $-$, can all be defined using the lifting machinery we have developed:

instance *matrix* :: (*plus*) *plus*
instance *matrix* :: (*minus*) *minus*
instance *matrix* :: ({*plus, times*}) *times*
defs (overloaded)

$$\begin{array}{rcl} A+B & \equiv & \textit{combine-matrix}(\lambda a\, b.\, a+b)\, A\, B \\ A-B & \equiv & \textit{combine-matrix}(\lambda a\, b.\, a-b)\, A\, B \\ -A & \equiv & \textit{apply-matrix}(\lambda a.\, -a)\, A \\ A*B & \equiv & \textit{mult-matrix}(\lambda a\, b.\, a*b)(\lambda a\, b.\, a+b)\, A\, B\ . \end{array} \tag{3.63}$$

Finally, we need to be able to compare matrices:

instance *matrix* :: ({*ord, zero*}) *ord*
defs (overloaded)

$$A \le B \quad \equiv \quad \forall\, j\, i.\, \textit{Rep-matrix}\, A\, j\, i \le \textit{Rep-matrix}\, B\, j\, i \tag{3.64}$$

After having introduced the necessary syntax, we need to show that α *matrix* really constitutes a lattice-ordered ring, provided α constitutes one, in order to obtain (3.60). But almost the entire work has already been done: for example, in order to prove associativity of matrix multiplication,

$$\forall (A :: (\alpha :: \textit{lordered-ring})\, \textit{matrix}).\, A*(B*C) = (A*B)*C\ , \tag{3.65}$$

which is the hardest of all proof obligations, just apply (3.56)! The remaining proof obligations are not difficult to prove, either, one just has to make use of matrix extensionality (3.25) and the lifting properties (3.40), (3.43) and (3.52). It is useful, though, first to dispose of the assumptions in these lifting properties, so for example instead of using (3.43) directly one should prove and use

$$\begin{array}{rcl} \textit{Rep-matrix}(A+B)\, j\, i & = & (\textit{Rep-matrix}\, A\, j\, i) + (\textit{Rep-matrix}\, B\, j\, i) \\ \textit{Rep-matrix}(A-B)\, j\, i & = & (\textit{Rep-matrix}\, A\, j\, i) - (\textit{Rep-matrix}\, B\, j\, i)\ . \end{array} \tag{3.66}$$

A proof obligation that differs from the others because it is not a universal property that needs to be shown, but an existential one, turns up when one has to show that *join* and *meet* do exist:

$$\begin{array}{l} \exists\, j.\, \forall\, a\, b\, x.\, a \le j\, a\, b \wedge b \le j\, a\, b \wedge (a \le x \wedge b \le x \longrightarrow j\, a\, b \le x) \\ \exists\, m.\, \forall\, a\, b\, x.\, m\, a\, b \le a \wedge m\, a\, b \le b \wedge (x \le a \wedge x \le b \longrightarrow x \le m\, a\, b) \end{array} \tag{3.67}$$

But these are not difficult to exhibit! Just choose

$$\textit{join} \equiv \textit{combine-matrix join},\ \ \textit{meet} \equiv \textit{combine-matrix meet}\ . \tag{3.68}$$

3.3.6 Positive Part and Negative Part

In lattice-ordered rings (actually in groups, also), both the positive part and the negative part can be defined:

constdefs

$$\begin{array}{l} \textit{pprt} :: \alpha \Rightarrow (\alpha :: \textit{lordered-ring}) \\ \textit{pprt}\, x \equiv \textit{join}\, x\, 0 \\ \textit{nprt} :: \alpha \Rightarrow (\alpha :: \textit{lordered-ring}) \\ \textit{nprt}\, x \equiv \textit{meet}\, x\, 0 \end{array} \tag{3.69}$$

We will write x^+ instead of *pprt x*, and x^- instead of *nprt x*. We have:

$$0 \le x^+, \quad x^- \le 0, \quad x = x^+ + x^-, \quad x \le y \Longrightarrow x^- \le y^- \wedge x^+ \le y^+ \ . \tag{3.70}$$

Positive part and negative part come in handy for calculating bounds for a product when bounds for each of the factors of the product are known:

$$\begin{aligned}&[\![a_1 \le a; a \le a_2; b_1 \le b; b \le b_2\ ]\!]\\ &\Longrightarrow a * b \le a_2^+ * b_2^+ + a_1^+ * b_2^- + a_2^- * b_1^+ + a_1^- * b_1^-\end{aligned} \tag{3.71}$$

In order to prove (3.71), decompose the factors into their parts and use distributivity. Then take advantage of the monotonicity of positive and negative part:

$$\begin{aligned}a * b &= (a^+ + a^-) * (b^+ + b^-)\\ &= a^+ * b^+ + a^+ * b^- + a^- * b^+ + a^- * b^-\\ &\le a_2^+ * b_2^+ + a_1^+ * b_2^- + a_2^- * b_1^+ + a_1^- * b_1^- \ .\end{aligned}$$

3.4 Proving Bounds by Duality

Now we have everything in place to represent LPs by finite matrices. In sect. 3.2, we presented the basic idea of how to prove an arbitrarily precise upper bound for the objective function (3.1) of a given LP. There the LP was represented by matrices whose elements are real numbers:

$$c \in \mathbb{R}^{1\times n}, \quad A \in \mathbb{R}^{m\times n}, \quad b \in \mathbb{R}^{m\times 1} \quad , \quad l, u \in \mathbb{R}^{n\times 1}.$$

Dropping the dimensions we arrive at a representation of a real linear program by finite matrices:

$$c, A, b, l, u :: \mathit{real\ matrix} \ .$$

From now on we are always talking in terms of finite matrices.

We need a further modification of our representation of LPs: our method is based on numerical algorithms like the Simplex method, therefore we need to represent the data numerically. We allow for this possibility by looking at *intervals* of linear programs instead of only considering a single LP. Such an interval is given by finite matrices c_1, c_2, A_1, A_2, b, l, u. We can now state the main theorem as it has been proven in Isabelle/HOL:

$$\begin{aligned}&[\![A * x \le b; A_1 \le A; A \le A_2; c_1 \le c; c \le c_2; l \le x; x \le u; 0 \le y]\!]\\ &\Longrightarrow c * x \le y * b + (\ \text{let } s_1 = c_1 - y * A_2; s_2 = c_2 - y * A_1\\ &\qquad\qquad \text{in } s_2^+ * u^+ + s_1^+ * u^- + s_2^- * l^+ + s_1^- * l^-)\ .\end{aligned} \tag{3.72}$$

The proof is by standard algebraic manipulations: using $A * x \le b$ and $y \ge 0$,

$$c * x \ \le \ y * b + (c - y * A) * x$$

follows at once. Then one just has to apply (3.71) to the product $(c - y * A) * x$. Note that this proof not only works for matrices, but for any lattice-ordered ring. Therefore the main theorem is valid also for lattice-ordered rings!

This is how our method works: First, we calculate the approximate optimal solution y of the dual LP. We know our primal LP only approximately, so we can pass only approximate data to the external LP solver. We could pass for example c_1,

A_1, b, l, u. The LP solver will return the certificate y, which is only approximately non-negative. Therefore we replace all negative elements of y by 0. We then plug the known numerical data $y, c_1, c_2, A_1, A_2, b, l$ and u into (3.72) and simplify the resulting theorem. The simplification will rewrite $0 \leq y$ to *True* and the large expression on the right hand side of the inequality to a matrix numeral K with $\mathit{ncols}\,K \leq 1$ and $\mathit{nrows}\,K \leq 1$. The result of our method is therefore the theorem

$$\begin{aligned} &[\![\mathbf{A} * \mathbf{x} \leq b; A_1 \leq \mathbf{A}; \mathbf{A} \leq A_2; c_1 \leq \mathbf{c}; \mathbf{c} \leq c_2; l \leq \mathbf{x}; \mathbf{x} \leq u]\!] \\ &\Longrightarrow \mathbf{c} * \mathbf{x} \leq K \ . \end{aligned} \tag{3.73}$$

In the above theorem, free variables are set in **bold face**. All other identifiers denote matrix numerals.

3.5 Proving Infeasibility by Modified Duality

If our linear program at hand is infeasible, the method of Section 3.4 will fail because the dual program will also be infeasible and the external solver will fail. How do we prove a bound of an infeasible linear program then? We have outlined the basic idea of how to do this already in Section 3.2.1. We take the infeasible linear program and modify it so that we get a new, feasible linear program. We then apply the method from Section 3.4 to bound this new linear program. Then we somehow find a way to relate the found bound of the new linear program to a bound for the original one according to Theorem 3.2.

Our original linear program is:

$$\text{Maximize } c * x \text{ where } x \text{ is subject to the condition } A * x \leq b.$$

For any K the modified linear program is to maximize $c * x + K * t$ subject to the condition $A * x + b * t \leq b$ and $0 \leq t \leq 1$. We can squeeze the modified linear program into standard form so that it becomes:

$$\text{Maximize } c' * x' \text{ where } x' \text{ is subject to the condition } A' * x' \leq b',$$

where

$$c' = \begin{pmatrix} c & K \end{pmatrix}, \quad x' = \begin{pmatrix} x \\ t \end{pmatrix}, \quad A' = \begin{pmatrix} A & b \\ 0 & 1 \\ 0 & -1 \end{pmatrix}, \quad b' = \begin{pmatrix} b \\ 1 \\ 0 \end{pmatrix}.$$

Now let's do for this modified linear program what we always do for a feasible linear program: solve the dual linear program and get a certificate y'. Because A' has two more rows than A, we can write y' as $y' = \begin{pmatrix} y & y_1 & y_2 \end{pmatrix}$ for some vector $y \geq 0$ and numbers $y_1 \geq 0$, $y_2 \geq 0$. If the original LP is infeasible, Theorem 3.2 implies

$$c' * x' \leq y' * b' \approx K$$

As this holds for any x', we can just as well choose an $x' = \begin{pmatrix} x \\ t \end{pmatrix}$ such that $t = 0$:

$$c * x = c' * \begin{pmatrix} x \\ 0 \end{pmatrix} \leq y' * b' = \begin{pmatrix} y & y_1 & y_2 \end{pmatrix} * \begin{pmatrix} b \\ 1 \\ 0 \end{pmatrix} = y * b + y_1 * 1 + y_2 * 0 = y * b + y_1 \approx K.$$

This looks terrific, because it bounds the objective function $c*x$ of the original linear program by $y*b+y_1$ which depends on the certificate of the modified linear program! Furthermore, because K was chosen arbitrarily, we can push the bound for $c*x$ as low as we want to. And the best thing is that we do not even need to prove any fancy new theorem for using this fact! The theorem

$$\begin{array}{l} [\![A*x \le b; A_1 \le A; A \le A_2; c_1 \le c; c \le c_2; l \le x; x \le u; 0 \le y; 0 \le y_1]\!] \\ \Longrightarrow c*x \le y*b+y_1+(\text{ let } s_1 = c_1 - y*A_2; s_2 = c_2 - y*A_1 \\ \qquad\qquad \text{in } s_2^+ * u^+ + s_1^+ * u^- + s_2^- * l^+ + s_1^- * l^-) \ . \end{array} \tag{3.74}$$

is just what we need and is because of $y_1 \ge 0$ a direct consequence of 3.72. To validate that this is the theorem we are looking for, we need to assure ourselves that

$$c - y*A \approx 0 \tag{3.75}$$

holds. Because y' is a certificate for the modified linear program we have

$$0 \approx c' - y' * A' = \begin{pmatrix} c & K \end{pmatrix} - \begin{pmatrix} y & y_1 & y_2 \end{pmatrix} * \begin{pmatrix} A & b \\ 0 & 1 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} c - y*A & K - (y*b + y_1 - y_2) \end{pmatrix}$$

Focusing on the first component in the above confirms (3.75), and therefore also the usefulness of (3.74).

We know now how to prove any bound we desire for any objective function $c*x$ of the original infeasible linear program. Actually, we get a particularly nice objective function by choosing $c = 0$. Specializing theorem (3.74) to $c = 0$ results in:

$$\begin{array}{l} [\![A*x \le b; A_1 \le A; A \le A_2; l \le x; x \le u; 0 \le y; 0 \le y_1]\!] \\ \Longrightarrow 0 \le y*b+y_1+(\text{ let } s_1 = -y*A_2; s_2 = -y*A_1 \\ \qquad\qquad \text{in } s_2^+ * u^+ + s_1^+ * u^- + s_2^- * l^+ + s_1^- * l^-) \ . \end{array} \tag{3.76}$$

Choosing a modification parameter K such that $K < 0$, for example $K = -1$, calculating a certificate y, y_1 for this modified LP, plugging this certificate into (3.76), and computing it will lead to the following theorem:

$$\begin{array}{l} [\![\mathbf{A}*\mathbf{x} \le b; A_1 \le \mathbf{A}; \mathbf{A} \le A_2; l \le \mathbf{x}; \mathbf{x} \le u]\!] \\ \Longrightarrow \mathit{False} \end{array} \tag{3.77}$$

We have arrived at an Isabelle theorem expressing the infeasibility of the original linear program. Free variables of the theorem are set in **bold face** again.

3.6 Sparse Matrices

After reading the previous sections, you probably wonder what a matrix numeral might look like. We have chosen to represent matrix numerals in such a way that sparse matrices are encoded efficiently:

$$\begin{array}{l} \textbf{types} \\ \quad \alpha\ \mathit{spvec} = (\mathit{nat} * \alpha)\ \mathit{list} \\ \quad \alpha\ \mathit{spmat} = (\alpha\ \mathit{spvec})\ \mathit{spvec} \end{array} \tag{3.78}$$

$$\begin{array}{l} \textbf{constdefs} \\ \quad \mathit{sparse\text{-}row\text{-}vector} :: \alpha\ \mathit{spvec} \Rightarrow \alpha\ \mathit{matrix} \\ \quad \mathit{sparse\text{-}row\text{-}vector}\ l \equiv \mathit{foldl}(\lambda\, m\, (i,e).\, m + (\mathit{singleton\text{-}matrix}\, 0\, i\, e))\, 0\, l \\ \quad \mathit{sparse\text{-}row\text{-}matrix} :: \alpha\ \mathit{spmat} \Rightarrow \alpha\ \mathit{matrix} \\ \quad \mathit{sparse\text{-}row\text{-}matrix}\ L \equiv \\ \qquad \mathit{foldl}(\lambda\, m\, (j,l).\, m + (\mathit{move\text{-}matrix}(\mathit{sparse\text{-}row\text{-}vector}\, l)\, j\, 0))\, 0\, L \end{array} \tag{3.79}$$

Here *singleton-matrix* $j\,i\,e$ denotes the matrix whose elements are all zero except the element in row j and column i, which equals e. Furthermore *move-matrix* $A\,j\,i$ denotes the matrix that one gets if one moves the matrix A by j rows down and i columns right, and fills up the first j rows and i columns with zero elements.

Here is an example of a matrix numeral:

$$[(1,[(1,7),(3,13)]),(2,[(0,-4),(1,47)])]$$

$$\xrightarrow{\textit{sparse-row-matrix}} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & 13 \\ -4 & 47 & 0 & 0 \end{pmatrix}$$

We have formalized addition, subtraction, multiplication, comparison, positive part and negative part directly on sparse vectors and matrices by recursion on lists. The multiplication algorithm for sparse matrices is inspired by the one given in [10].

These operations on sparse vectors/matrices can be proven correct with respect to their finite matrices counterpart via the *sparse-row-matrix* morphism, assuming certain sortedness constraints. This is actually not too hard: all students of an introductory Isabelle/HOL class taught at Technische Universität München have been able to complete these proofs within four weeks as their final assignment with varying help from their tutors. Using these correctness results, one can then easily prove a sparse version of (3.72).

3.7 Interval Arithmetic

3.7.1 Floats

Real numbers are represented as binary, arbitrary precision floating point numbers:

constdef
$$\begin{aligned} &\textit{float} :: (\textit{int} * \textit{int}) \Rightarrow \textit{real} \\ &\textit{float}\,(m,e) \equiv (\textit{real}\; m) * 2^e \end{aligned} \tag{3.80}$$

In the above, m is called the *mantissa*, and e the *exponent*. The Isabelle theorem

$$\begin{aligned} &\textit{float}\;(a_1, e_1) + \textit{float}\;(a_2, e_2) = \\ &\quad \textit{if}\; e_1 \le e_2\; \textit{then float}\;(a_1 + a_2 * 2 \,\hat{}\, \textit{nat}\;(e_2 - e_1), e_1) \\ &\quad \textit{else float}\;(a_1 * 2 \,\hat{}\, \textit{nat}\;(e_1 - e_2) + a_2, e_2) \end{aligned} \tag{3.81}$$

shows how to perform addition in this representation, multiplication is even easier:

$$\textit{float}\;(a_1, e_1) * \textit{float}\;(a_2, e_2) = \textit{float}\;(a_1 * a_2, e_1 + e_2) \tag{3.82}$$

We prefer a floating point representation over a representation by fractions because addition of floating point numbers is cheaper and easier to perform than addition of fractions. We will often want to multiply matrices with real numbers as entries, and there addition and multiplication of real numbers are the dominant operations.

Other operations like the positive and the negative part are also easily definable for floats. Division is problematic, though. Take for example the real number expressed by the fraction $\frac{1}{3}$. There are no m,e such that $\textit{float}\,(m,e) = \frac{1}{3}$. But because our floating point representation is arbitrary precision, it is of course no problem to approximate $\frac{1}{3}$ by floats as precise as we wish for. E.g., up to a precision of 2^{-50}:

$$\mathit{float}\,(375299968947541, -50) \le \tfrac{1}{3} \le \mathit{float}\,(750599937895083, -51)$$

The failure of floats to represent certain fractions directly is no reason to abandon the float representation and to prefer fractions instead. When we later turn to the basic linear programs we will have to compute with values like π or $\sqrt{2}$ which even fractions cannot represent directly, therefore approximation is inevitable.

We want to develop algorithms for floats, e.g. division, and we want these algorithms to be functions *within the logic*, so that we can apply the HOL Computing Library to it. These algorithms need to be able to break a float into its components mantissa and exponent. Because $\mathit{float}(m,e)$ is just a real number, and there are no such things as the mantissa and exponent of a real number, we need to introduce a new type of floats.

datatype *float = Float int int*

fun *Ifloat* :: *float* ⇒ *real* (3.83)
where
Ifloat (*Float a b*) = *float* (*a*,*b*)

Operations like addition, multiplication and so on can be defined on this type. All of these operations must commute with the *Ifloat* morphism. One can show that the type *float* is an instance of the axiomatic type class of commutative semirings. It is not the case that *float* is an instance of the type class *order*, because then it would have to fulfill $(x < y) = (x \le y \land x \ne y)$ which is clearly not the case.

3.7.2 Division of Floats

We approach the division algorithm in four steps. We first approximate nonnegative true fractions like $\frac{1}{3}$ or $\frac{4}{5}$. We then move on to approximating any nonnegative fraction, like $\frac{10}{7}$, and finally to approximating any fraction, including negative ones. From there it is straightforward to define division of floats.

The *lapprox-frac* and the *rapprox-frac* functions approximate a nonnegative true fraction from the left and the right, respectively. The quality of the approximation can be controlled by a counter that determines the number of approximation steps. By a flag one can stop this counter from counting until the significant digits of the number have been reached.

function *lapprox-frac* :: *bool* ⇒ *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
lapprox-frac flag 0 x y = 0
| *x* ≤ *0* ⟹ *lapprox-frac flag n x y = 0*
| *0 < x* ⟹ *0 < n* ⟹ (3.84)
lapprox-frac flag n x y =
(*if 2∗x ≥ y then Float 1 −1 + Float 1 −1 ∗ lapprox-frac True* (*n* − *1*) (*2∗x−y*) *y*
else Float 1 −1 ∗ lapprox-frac flag (*if flag then* (*n* − *1*) *else n*) (*2∗x*) *y*)

function *rapprox-frac* :: *bool* ⇒ *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
rapprox-frac flag 0 x y = (*if x ≤ 0 then 0 else 1*)
| *x* ≤ *0* ⟹ *rapprox-frac flag n x y = 0*
| *0 < x* ⟹ *0 < n* ⟹ (3.85)
rapprox-frac flag n x y =
(*if 2∗x ≥ y then Float 1 −1 + Float 1 −1 ∗ rapprox-frac True* (*n* − *1*) (*2∗x−y*) *y*
else Float 1 −1 ∗ rapprox-frac flag (*if flag then* (*n* − *1*) *else n*) (*2∗x*) *y*)

In order to define these functions one has to prove termination, which was surprisingly difficult, involving over 150 lines of proof.

The correctness of *lapprox-frac* and *rapprox-frac* is stated in (3.86) and (3.87).

$$0 \le x \Longrightarrow x < y \Longrightarrow 0 \le \mathit{Ifloat}\ (\mathit{lapprox\text{-}frac}\ \mathit{flag}\ n\ x\ y) \wedge \mathit{Ifloat}\ (\mathit{lapprox\text{-}frac}\ \mathit{flag}\ n\ x\ y) \le \mathit{real}\ x\ /\ \mathit{real}\ y \tag{3.86}$$

$$0 \le x \Longrightarrow x < y \Longrightarrow \mathit{real}\ x\ /\ \mathit{real}\ y \le \mathit{Ifloat}\ (\mathit{rapprox\text{-}frac}\ \mathit{flag}\ n\ x\ y) \tag{3.87}$$

Note that we did not prove any theorems about the *quality* of the approximation, which are not needed for our formal development, but tests show it to be as expected.

We will now deal with *any* nonnegative fraction:

definition *lapprox-posrat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
lapprox-posrat prec x y =
(*let*
d = *x div y*;
m = *x mod y*
in
Float d 0 + *lapprox-frac* (*d* ≠ *0*) (*prec* − *nat* (*bitlen d*)) *m y*) (3.88)

definition *rapprox-posrat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
rapprox-posrat prec x y =
(*let*
d = *x div y*;
m = *x mod y*
in
Float d 0 + *rapprox-frac* (*d* ≠ *0*) (*prec* − *nat* (*bitlen d*)) *m y*) (3.89)

The function *bitlen* counts the number of bits of its argument. The correctness results for *lapprox-posrat* and *rapprox-posrat* are similar to (3.86) and (3.87), but with the condition $x < y$ weakened to $0 < y$.

Approximating *any* fraction is now just a matter of case distinction:

function *lapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
y = *0* ⟹ *lapprox-rat prec x y* = *0*
| *0* ≤ *x* ⟹ *0* < *y* ⟹ *lapprox-rat prec x y* = *lapprox-posrat prec x y* (3.90)
| *x* < *0* ⟹ *0* < *y* ⟹ *lapprox-rat prec x y* = − (*rapprox-posrat prec* (−*x*) *y*)
| *x* < *0* ⟹ *y* < *0* ⟹ *lapprox-rat prec x y* = *lapprox-posrat prec* (−*x*) (−*y*)
| *0* ≤ *x* ⟹ *y* < *0* ⟹ *lapprox-rat prec x y* = − (*rapprox-posrat prec x* (−*y*))

function *rapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
y = *0* ⟹ *rapprox-rat prec x y* = *0*
| *0* ≤ *x* ⟹ *0* < *y* ⟹ *rapprox-rat prec x y* = *rapprox-posrat prec x y* (3.91)
| *x* < *0* ⟹ *0* < *y* ⟹ *rapprox-rat prec x y* = − (*lapprox-posrat prec* (−*x*) *y*)
| *x* < *0* ⟹ *y* < *0* ⟹ *rapprox-rat prec x y* = *rapprox-posrat prec* (−*x*) (−*y*)
| *0* ≤ *x* ⟹ *y* < *0* ⟹ *rapprox-rat prec x y* = − (*lapprox-posrat prec x* (−*y*))

Here are the theorems about the correctness of these functions:

$$\mathit{Ifloat}\ (\mathit{lapprox\text{-}rat}\ \mathit{prec}\ x\ y) \le \mathit{real}\ x\ /\ \mathit{real}\ y \tag{3.92}$$

$$\textit{real } x \;/\; \textit{real } y \le \textit{Ifloat } (\textit{rapprox-rat prec } x\ y) \tag{3.93}$$

Finally, we can compute the division of floats:

fun *float-divl* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float*
where
float-divl prec (*Float* m_1 s_1) (*Float* m_2 s_2) =
(*let*
l = *lapprox-rat prec* m_1 m_2;
f = *Float 1* ($s_1 - s_2$)
in
f ∗ *l*) (3.94)

fun *float-divr* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float*
where
float-divr prec (*Float* m_1 s_1) (*Float* m_2 s_2) =
(*let*
r = *rapprox-rat prec* m_1 m_2;
f = *Float 1* ($s_1 - s_2$)
in
f ∗ *r*) (3.95)

The correctness is an immediate corollary of (3.92) and (3.93), respectively:

$$\textit{Ifloat } (\textit{float-divl prec } x\ y) \le \textit{Ifloat } x \;/\; \textit{Ifloat } y \tag{3.96}$$

$$\textit{Ifloat } x \;/\; \textit{Ifloat } y \le \textit{Ifloat } (\textit{float-divr prec } x\ y) \tag{3.97}$$

3.7.3 Basic Interval Arithmetic for Floats

We have seen how to formalize addition, multiplication and division for floats. Addition and multiplication commute with the *Ifloat* morphism; by that we mean that the formulas

$$\textit{Ifloat } (u+v) = \textit{Ifloat } u + \textit{Ifloat } v, \qquad \textit{Ifloat } (u*v) = \textit{Ifloat } u * \textit{Ifloat } v \tag{3.98}$$

are true for all u, v of type *float*. As we have seen, this is not true for division; there is no function d on floats with the property

$$\textit{Ifloat } (d\ u\ v) = \textit{Ifloat } u / \textit{Ifloat } v \tag{3.99}$$

for all u, v of type *float*. Also, we might be interested in different addition and multiplication operations than those exact ones with the commute property. E.g., imagine that throughout a computation we do not want to use more than 30 binary digits.

The commute property is especially nice when evaluating larger expressions which consist of nested elementary operations. Let's say we have four real numbers x_1, x_2, x_3, x_4 together with their float representations $x_i = \textit{Ifloat } r_i$. Then calculating $x_1 + (x_2 * (x_3 + x_4))$ is just a matter of simple rewriting:

$$\begin{aligned} x_1 + (x_2 * (x_3 + x_4)) &= \textit{Ifloat } r_1 + (\textit{Ifloat } r_2 * (\textit{Ifloat } r_3 + \textit{Ifloat } r_4) \\ &= \textit{Ifloat } (r_1 + (r_2 * (r_3 + r_4))) \\ &= \textit{float } (a, b) \end{aligned} \tag{3.100}$$

where $r_1 + (r_2 * (r_3 + r_4))$ rewrites to *Float a b*.

Losing the commute property means losing this simple way of evaluating nested operations by rewriting, because now the relationship between two steps in the computation is not equality any more, but something more complicated.

We do not want to give up the speed of rewriting with the HOL Computing Library; fortunately, there is a solution to our problem. We introduce a new datatype which represents nested expressions of basic operations:

datatype *basicarith* =
Plus basicarith basicarith | *Sub basicarith basicarith* | *Minus basicarith*
| *Mult basicarith basicarith* | *Div basicarith basicarith* | *Inverse basicarith*
| *Atom nat* | *Num float* (3.101)

The *Atom* constructor acts as a variable in de-Bruijn representation, so that atomic values which do not correspond to any other *basicarith* constructor can nevertheless be incorporated into the expression. In order to assign meaning to a *basicarith* expression we need to bind those variables to values. Therefore the *Ibasicarith* function not only takes a *basicarith* term as an argument, but also an environment of real numbers which is supposed to bind any variable occurring in the term.

fun *Ibasicarith* :: *real list* ⇒ *basicarith* ⇒ *real*
where
Ibasicarith vs (*Plus a b*) = (*Ibasicarith vs a*) + (*Ibasicarith vs b*)
| *Ibasicarith vs* (*Sub a b*) = (*Ibasicarith vs a*) − (*Ibasicarith vs b*)
| *Ibasicarith vs* (*Minus a*) = − (*Ibasicarith vs a*)
| *Ibasicarith vs* (*Mult a b*) = (*Ibasicarith vs a*) ∗ (*Ibasicarith vs b*) (3.102)
| *Ibasicarith vs* (*Div a b*) = (*Ibasicarith vs a*) / (*Ibasicarith vs b*)
| *Ibasicarith vs* (*Inverse a*) = *1* / (*Ibasicarith vs a*)
| *Ibasicarith vs* (*Num f*) = *Ifloat f*
| *Ibasicarith vs* (*Atom n*) = *vs* ! *n*

The expression $vs!n$ picks the n-th element out of the list *vs*.

One can understand the *basicarith* type as an extension of the *float* type such that every operation on the real numbers we are interested in has a corresponding commuting operation.

How do we use the *Ibasicarith* morphism for evaluating nested expressions? Look at the expression

$$\frac{\pi+4}{\sqrt{2}}.$$

There are two values in this expression, π and $\sqrt{2}$, which cannot be represented directly as *basicarith* terms. We factor them out and turn them into variables. This gives us:

$$\frac{\pi+4}{\sqrt{2}} = \mathit{Ibasicarith}\ [\pi,\ \sqrt{2}]\ (\mathit{Div}\ (\mathit{Add}\ (\mathit{Atom}\ 0)(\mathit{Num}\ (\mathit{Float}\ 1\ 2)))\ (\mathit{Atom}\ 1)). \quad (3.103)$$

We want to approximate the right hand side using interval arithmetic. We define an approximation function on *basicarith*:

$$
\begin{aligned}
&\textbf{function}\ \mathit{approx} :: \mathit{nat} \Rightarrow (\mathit{float}{*}\mathit{float})\ \mathit{list} \Rightarrow \mathit{basicarith} \Rightarrow (\mathit{float}{*}\mathit{float})\ \mathit{option}\\
&\textbf{where}\\
&\ \mathit{approx}\ n\ \mathit{bs}\ (\mathit{Atom}\ i) = (\mathit{if}\ i < \mathit{length}\ \mathit{bs}\ \mathit{then}\ \mathit{Some}\ (\mathit{bs}\ !\ i)\ \mathit{else}\ \mathit{None})\\
&|\ \mathit{approx}\ n\ \mathit{bs}\ (\mathit{Num}\ f) = \mathit{Some}\ (f, f)\\
&|\ \mathit{approx}\ n\ \mathit{bs}\ (\mathit{Plus}\ a\ b) =\\
&\qquad \mathit{lift\text{-}bin}\ (\mathit{approx}\ n\ \mathit{bs}\ a)\ (\mathit{approx}\ n\ \mathit{bs}\ b)\\
&\qquad\ (\lambda\ a_1\ a_2\ b_1\ b_2.\ \mathit{Some}\ (a_1 + b_1, a_2 + b_2))\\
&|\ \mathit{approx}\ n\ \mathit{bs}\ (\mathit{Sub}\ a\ b) =\\
&\qquad \mathit{lift\text{-}bin}\ (\mathit{approx}\ n\ \mathit{bs}\ a)\ (\mathit{approx}\ n\ \mathit{bs}\ b)\\
&\qquad\ (\lambda\ a_1\ a_2\ b_1\ b_2.\ \mathit{Some}\ (a_1 - b_2, a_2 - b_1))\\
&|\ \mathit{approx}\ n\ \mathit{bs}\ (\mathit{Mult}\ a\ b) =\\
&\qquad \mathit{lift\text{-}bin}\ (\mathit{approx}\ n\ \mathit{bs}\ a)\ (\mathit{approx}\ n\ \mathit{bs}\ b)\\
&\qquad\ (\lambda\ a_1\ a_2\ b_1\ b_2.\ \mathit{Some}\\
&\qquad\quad (\mathit{float\text{-}nprt}\ a_1 * \mathit{float\text{-}pprt}\ b_2 + \mathit{float\text{-}nprt}\ a_2 * \mathit{float\text{-}nprt}\ b_2\\
&\qquad\quad + \mathit{float\text{-}pprt}\ a_1 * \mathit{float\text{-}pprt}\ b_1 + \mathit{float\text{-}pprt}\ a_2 * \mathit{float\text{-}nprt}\ b_1,\\
&\qquad\quad \mathit{float\text{-}pprt}\ a_2 * \mathit{float\text{-}pprt}\ b_2 + \mathit{float\text{-}pprt}\ a_1 * \mathit{float\text{-}nprt}\ b_2\\
&\qquad\quad + \mathit{float\text{-}nprt}\ a_2 * \mathit{float\text{-}pprt}\ b_1 + \mathit{float\text{-}nprt}\ a_1 * \mathit{float\text{-}nprt}\ b_1))\\
&|\ \mathit{approx}\ n\ \mathit{bs}\ (\mathit{Inverse}\ a) =\\
&\qquad \mathit{lift\text{-}un}\ (\mathit{approx}\ n\ \mathit{bs}\ a)\\
&\qquad\ (\lambda\ a_1\ a_2.\ \mathit{if}\ (0 < a_1 \vee a_2 < 0)\ \mathit{then}\\
&\qquad\qquad\qquad \mathit{Some}\ (\mathit{float\text{-}divl}\ n\ 1\ a_2, \mathit{float\text{-}divr}\ n\ 1\ a_1)\\
&\qquad\qquad\quad \mathit{else}\ \mathit{None})\\
&|\ \mathit{approx}\ n\ \mathit{bs}\ (\mathit{Div}\ a\ b) = \mathit{approx}\ n\ \mathit{bs}\ (\mathit{Mult}\ a\ (\mathit{Inverse}\ b))\\
&|\ \mathit{approx}\ n\ \mathit{bs}\ (\mathit{Minus}\ a) = \mathit{lift\text{-}un}\ (\mathit{approx}\ n\ \mathit{bs}\ a)\ (\lambda\ a_1\ a_2.\ \mathit{Some}\ (-\ a_2, -\ a_1))
\end{aligned}
\tag{3.104}
$$

Given an environment *bs* that consists of intervals of floats for each bound variable, *approx prec bs t* will return an interval of floats that approximates t. It might not be able to compute such an interval; then it returns indicating a failure. If the term makes no use of division or inversion, the returned interval will be the tightest one possible. Use of division or inversion introduces approximation; the quality of this approximation is controlled by the *prec* parameter. The algorithm should be fairly self-explaining. Because each recursive call to *approx* may fail (inversion of an interval containing 0 is not possible), we use functions *lift-un*(ary) and *lift-bin*(ary) which either propagate the failure, or continue processing in case of success. We deal with the case for multiplication using our formula for estimating a product by the positive and negative part of its components (3.71). Division is delegated to inversion, and inversion is dealt with using the algorithms for division of floats from Section 3.7.2.

To express the correctness of *approx* we need to define a predicate which says if a list of real numbers is bounded by a list of intervals:

$$
\begin{aligned}
&\textbf{definition}\\
&\ \mathit{bounded\text{-}by} :: \mathit{real}\ \mathit{list} \Rightarrow (\mathit{float} * \mathit{float})\ \mathit{list} \Rightarrow \mathit{bool}\\
&\textbf{where}\\
&\ \mathit{bounded\text{-}by}\ \mathit{vs}\ \mathit{bs} =\\
&\qquad (\forall\ i.\ i < \mathit{length}\ \mathit{bs} \longrightarrow \mathit{Ifloat}\ (\mathit{fst}\ (\mathit{bs}\ !\ i)) \le \mathit{vs}\ !\ i \wedge \mathit{vs}\ !\ i \le \mathit{Ifloat}\ (\mathit{snd}\ (\mathit{bs}\ !\ i)))
\end{aligned}
\tag{3.105}
$$

The correctness result for *approx* can then be stated like this:

$$
\begin{aligned}
&[\![\mathit{bounded\text{-}by}\ \mathit{vs}\ \mathit{bs};\ \mathit{approx}\ \mathit{prec}\ \mathit{bs}\ \mathit{expr} = \mathit{Some}\ (l, r)]\!] \Longrightarrow\\
&\ \mathit{Ifloat}\ l \le \mathit{Ibasicarith}\ \mathit{vs}\ \mathit{expr} \wedge \mathit{Ibasicarith}\ \mathit{vs}\ \mathit{expr} \le \mathit{Ifloat}\ r
\end{aligned}
\tag{3.106}
$$

Let us finish the example we started in (3.103). We turn bounds for π and $\sqrt{2}$

$$\textit{bounded-by}\ [\pi,\ \sqrt{2}]\ [(\textit{Float}\ 3\ 0,\ \textit{Float}\ 1\ 2), (\textit{Float}\ 1\ 0,\ \textit{Float}\ 1\ 1)] \tag{3.107}$$

into an approximation of $(\pi+4)/\sqrt{2}$:

$$\begin{aligned} &\textit{approx}\ 1\ [(\textit{Float}\ 3\ 0,\ \textit{Float}\ 1\ 2), (\textit{Float}\ 1\ 0,\ \textit{Float}\ 1\ 1)] \\ &\qquad (\textit{Div}\ (\textit{Add}\ (\textit{Atom}\ 0)(\textit{Num}\ (\textit{Float}\ 1\ 2))) \\ &= \textit{Some}\ (\textit{Float}\ 7\ -1,\ \textit{Float}\ 1\ 3) \end{aligned} \tag{3.108}$$

Plugging (3.107) and (3.108) into (3.106) and applying (3.103) yields the theorem

$$\textit{float}\ (7,\ -1) \le \frac{\pi+4}{\sqrt{2}} \quad \wedge \quad \frac{\pi+4}{\sqrt{2}} \le \textit{float}\ (1,\ 3). \tag{3.109}$$

3.7.4 Approximation of Matrices

Let us assume that the matrix A of a linear program is given by a matrix with elements like $\frac{\pi+4}{\sqrt{2}}$. We know how to approximate a single element of this matrix; in order to apply our methods for bounding linear programs, we need to approximate *all* entries of the matrix and obtain both a lower bound A_1 and an upper bound A_2 for A. We achieve this by lifting *approx* to the level of matrices. The signature for a version of *approx* for sparse matrices is:

$$\begin{aligned} &\textit{approx-spmat} :: \\ &\textit{nat} \Rightarrow (\textit{float}*\textit{float})\ \textit{list} \Rightarrow \textit{basicarith spmat} \Rightarrow (\textit{float spmat} * \textit{float spmat})\ \textit{option}. \end{aligned} \tag{3.110}$$

In our application there are many entries of the matrix which are equal. Therefore factoring out whole equal entries as variables and approximating them separately significantly improves the performance of the approximation.

3.8 Calculating A Priori Bounds

Both the method of proving bounds for feasible linear programs from Section 3.4 and the method of proving infeasibility from Section 3.5 require a priori knowledge of bounds for the vector x. We might be able to read particularly obvious bounds off the constraints $A * x \le b$. Consider for example the case that one row

$$a_j = \begin{pmatrix} a_{j,1} & \dots & a_{j,n} \end{pmatrix} \quad \text{of} \quad A = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}$$

has only one non-null entry, say $a_{j,i}$. Then either $x_i \le b_j / a_{j,i}$ if $a_{j,i} > 0$ or $x_i \ge b_j / a_{j,i}$ if $a_{j,i} < 0$.

There are more easy cases we can handle. Let us again single out the matrix entry $a_{j,i} \neq 0$ but this time we do not necessarily assume all other entries in the same row to be zero. We know $\sum_{k=1}^{n} a_{j,k}\, x_k \le b_j$, or, after moving all terms except $a_{j,i}\, x_i$ to the right hand side of the inequality,

$$a_{j,i}\, x_i \quad \le \quad b_j + \sum_{k \in \{1,\dots,n\},\ k \neq i} (-a_{j,k}\, x_k). \tag{3.111}$$

The term $-a_{j,k}\, x_k$ can be bounded from the above in one of these cases:

- $a_{j,k} = 0$,

- $a_{j,k} > 0$, and we already know a lower bound for x_k,
- $a_{j,k} < 0$, and we already know an upper bound for x_k.

If we can thus bound for all $k \neq i$ the terms $-a_{j,k}\, x_k$ then we can bound x_i from above if $a_{j,i} > 0$, or we can bound x_i from below if $a_{j,i} < 0$.

We repeat adding such easy bounds to our list of known bounds for x until the set of indices i for which we know a lower bound of x_i and the set of indices i for which we know an upper bound of x_i do not change any more. This computation is performed *outside the logic*. What we have got after finishing the computation is a *certificate* for proving formally within the logic bounds for the components of x. This certificate is just a list of triples (j, i, ks), where i denotes the row x_i we are going to bound, j denotes the row a_j and ks is the set of indices $k \neq i$ such that $a_{j,k} \neq 0$.

We work the certificate from its first to its last element. For a single element of the certificate we instantiate the following theorem:

$$\begin{array}{l} [\![A_1 \leq A;\ A \leq A_2;\ A * x \leq b; \\ \quad \textit{row-of-matrix}\ A_1\ j = a_1;\ \textit{row-of-matrix}\ A_2\ j = a_2; \\ \quad \textit{nullify-column}\ a_1\ i = u_1;\ \textit{nullify-column}\ a_2\ i = u_2; \\ \quad \textit{filter-cols}\ u_1\ ks = u_1;\ \textit{filter-cols}\ u_2\ ks = u_2; \\ \quad \textit{filter-rows}\ x\ ks = x';\ x_1 \leq x';\ x' \leq x_2; \\ \quad \textit{Rep-matrix}\ a_1\ 0\ i = \textit{float}\ (m_1, e_1);\ \textit{Rep-matrix}\ a_2\ 0\ i = \textit{float}\ (m_2, e_2); \\ \quad \textit{Rep-matrix}\ b\ j\ 0 - \textit{Rep-matrix}\ (u_1^- * x_2^+ + u_2^- * x_2^- + u_1^+ * x_1^+ + u_2^+ * x_1^-)\ 0\ 0 = \textit{float}\ (m, e); \\ \quad \textit{approx prec}\ [(\textit{Float}\ m_1\ e_1,\ \textit{Float}\ m_2\ e_2)]\ (\textit{Div}\ (\textit{Num}\ (\textit{Float}\ m\ e))\ (\textit{Atom}\ 0)) = \textit{Some}\ (\textit{lapprox}, \textit{rapprox})]\!] \\ \Longrightarrow \textbf{if}\ 0 < m_1\ \textbf{then}\ \textit{Rep-matrix}\ x\ i\ 0 \leq \textit{Ifloat rapprox} \\ \quad\ \textbf{else if}\ m_2 < 0\ \textbf{then}\ \textit{Ifloat lapprox} \leq \textit{Rep-matrix}\ x\ i\ 0 \\ \quad\ \textbf{else}\ \textit{True} \end{array} \tag{3.112}$$

The first line of (3.112) sets the context; we have a system of linear inequalities $A * x \leq b$, and the matrix A is approximated by matrices A_1 and A_2. The rest of the theorem can be read like a program with occasional assertions. We calculate the j-th row of A_1 and A_2 and store them in a_1 and a_2, respectively. Then the i-th column of a_1 and a_2 are set to 0, resulting in u_1 and u_2. Next we need to ensure that the set ks really contains all indices k for which the k-th column of a_j is non-zero. This is what *filter-cols* u_1 $ks = u_1$ and *filter-cols* u_2 $ks = u_2$ check; if we retrieve all columns of u_1 with indices in ks by setting all other columns to 0, and the result is u_1, then clearly setting the other columns to 0 has not changed anything, so they must have been 0 before. The same holds for u_2. The next line of the program, *filter-rows* x $ks = x'$, acknowledges the fact that only those rows of x with indices in ks are relevant for calculating the bound. We check then that these rows can be bounded by x_1 and x_2, where x_1 and x_2 have been obtained by the previous steps of working the certificate.

Next we store the lower and the upper bound for $a_{j,i}$ as *float* (m_1, e_1) and *float* (m_2, e_2). We then calculate an upper bound of the right hand side of Equation 3.111 and store the result as *float* (m,e). Dividing *float* (m,e) by $a_{j,i}$ gives us an upper bound for x_i if $a_{j,i} > 0$, and a lower bound if $a_{j,i} < 0$.

It would be nice if our theorem prover could understand (3.112) as a program with assertions as we just explained it. The HOL Computing Library allows the theorem prover to see them that way. In Section 2.6.4 we outlined the HCL's capabilities of mixing modus ponens, variable instantiation and computing. This is what is needed here. We first internalize Theorem (3.112) as HCL theorem and use modus ponens to discharge the first three assumptions, thereby setting the context. This gives us an HCL theorem Ψ. For each element $e = (j, i, ks)$ of the certificate we then instantiate the corresponding variables in Ψ, resulting in an HCL theorem Ψ_e. Eliminating all assumptions of Ψ_e and exporting the result gives us an Isabelle theorem that states a bound for x_i.

Finally, note that actually we do not use 3.112 directly but a version for sparse matrices derived from it.

C H A P T E R **4**

The Basic Linear Programs

Reality is that which, when you stop believing in it, doesn't go away.
— Philip K. Dick

Contents

4.1 The Archive Of Tame Graphs

The result of the first major completed part of the Flyspeck project is the verification of an *archive of tame graphs* [23]. Tame graphs are special planar graphs that represent possible counterexamples to the Kepler conjecture. There are only finitely many, and that is why it is possible to have an archive of all of them.

There are various ways of representing and formalizing planar graphs [3]. As a planar graph consists of nodes, edges and faces, one possibility to represent a planar graph would be a list of faces, where each face is again a list of nodes, leaving the representation of the edges implicit in the arrangement of nodes. This is the representation used in [23].

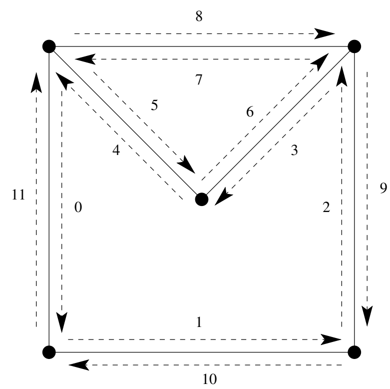
For our work we choose to represent planar graphs by *hypermaps*. Hypermaps have been introduced to mechanized theorem proving by Gonthier in his proof of the Four Color Theorem [8]. It builds on the notion of *dart*. Darts can be viewed as the oriented edges of a graph (Gonthier chooses a different, but equivalent, point of view; he sees them as angles between incident edges of the same face). Let us assume that we are given three permutations e, n and f of a finite set D of darts. Two darts α and β are called equivalent with respect to a permutation p of D iff $\exists n.\ \alpha = p^n \beta$ holds. The such induced relation on darts is an equivalence relation, and we write $|p|$ for its number of equivalence classes. If now the given three permutations fulfill the equations

$$e = e^{-1} = n \circ f \quad \text{and} \quad 2|e| = |D| \quad \text{and} \quad |e| + |f| + |n| = |D| + 2, \tag{4.1}$$

then we can construct a connected planar graph in the following way:

1. The nodes, edges and faces are the equivalence classes of n, e and f.
2. An edge $E = \{\delta_1, \delta_2\}$ consists therefore of the two oriented edges δ_1 and δ_2.
3. There is an oriented edge δ from node N_1 to node N_2 iff $\delta \in N_1$ and $e\delta \in N_2$.
4. An oriented edge δ belongs to a face F iff $\delta \in F$.

Figure 4.1: A planar graph



Consider the planar graph in Fig. 4.1. Each dart is denoted by a number, and we have $D = \{0, 1, 2, \ldots, 11\}$. The permutations are given by

$$\begin{array}{rcl} f &=& (0 \mapsto 1 \mapsto 2 \mapsto 3 \mapsto 4,\ 5 \mapsto 6 \mapsto 7,\ 8 \mapsto 9 \mapsto 10 \mapsto 11), \\ e &=& (0 \mapsto 11,\ 1 \mapsto 10,\ 2 \mapsto 9,\ 3 \mapsto 6,\ 4 \mapsto 5,\ 7 \mapsto 8), \\ n &=& (0 \mapsto 5 \mapsto 8,\ 1 \mapsto 11,\ 2 \mapsto 10,\ 3 \mapsto 9 \mapsto 7,\ 4 \mapsto 6). \end{array} \tag{4.2}$$

It might be instructive to check the equations (4.1).

Darts are also viewed as faces, nodes and edges. For example, there are three faces, face 0, face 5, and face 8 (to see why there are three faces, not two, imagine how the graph looks when drawn on a sphere instead of in the plane); and face 7 is the same as face 5.

We have converted the archive of tame graphs [22] into the hypermap representation and formalized them as values of type ($nat \times nat \times nat$) *NatTreeMap* where

datatype α *NatTreeMap* = *TIN nat* α (α *NatTreeMap*) (α *NatTreeMap*) | *TNN* (4.3)

An α *NatTreeMap* represents a function via the *eval* function:

fun *eval* :: α *NatTreeMap* $\Rightarrow$ *nat* $\Rightarrow$ α *option*
where
eval TNN x = None (4.4)
| *eval* (*TIN x c a b*) *x′* =
if x=x′ then Some c else if x′ < x then eval a x′ else eval b x′

The face, edge, and node permutations of a graph represented as value of type (*nat*×*nat*×*nat*) *NatTreeMap* can therefore be accessed through the following three functions:

$$\begin{array}{l}\textbf{constdefs}\\ \quad \textit{map-face} :: (\textit{nat} \times \textit{nat} \times \textit{nat})\ \textit{NatTreeMap} \Rightarrow \textit{nat} \Rightarrow \textit{nat}\\ \quad \textit{map-face}\ m\ d \equiv \textit{fst}\ (\textit{the}\ (\textit{eval}\ m\ d))\\ \quad \textit{map-edge} :: (\textit{nat} \times \textit{nat} \times \textit{nat})\ \textit{NatTreeMap} \Rightarrow \textit{nat} \Rightarrow \textit{nat}\\ \quad \textit{map-edge}\ m\ d \equiv \textit{fst}\ (\textit{snd}\ (\textit{the}\ (\textit{eval}\ m\ d)))\\ \quad \textit{map-node} :: (\textit{nat} \times \textit{nat} \times \textit{nat})\ \textit{NatTreeMap} \Rightarrow \textit{nat} \Rightarrow \textit{nat}\\ \quad \textit{map-node}\ m\ d \equiv \textit{snd}\ (\textit{snd}\ (\textit{the}\ (\textit{eval}\ m\ d)))\end{array} \tag{4.5}$$

The complete archive of 2771 tame graphs is given in our formalization as a constant *Archive* of type (*nat*×*nat*×*nat*) *NatTreeMap list*, each element of the list representing a tame graph. For convenience, there are also 2771 constants

$$\textit{graph-}1, \ldots, \textit{graph-}2771$$

of type (*nat*×*nat*×*nat*) *NatTreeMap*, each constant representing a tame graph.

4.2 Graph Systems

A *graph system* models a planar graph together with

- a fixed set of variables defined on the darts of the planar graphs,
- constraints relating these variables.

In [16, sec. 23.3] Hales sketches what he calls the *basic linear programs*. Our notion of graph system is intended to encompass those basic linear programs and give a complete specification of them. It *does not* capture the more complicated linear programs which result from branch-and-bound methods. In order to handle those linear programs, it would be necessary to add more variables to the graph system and to change the constraints on them.

Note that a graph system is a way to ascribe certain properties to a given planar graph. We do not intend the graph system to model the notion of planarity exactly; for example, we do not require the graph induced by a graph system to fulfill all of the properties in (4.1). But for the concrete instances of graph systems we are dealing with these equations will hold, of course, because these concrete instances are based on tame graphs, and tame graphs are planar.

We manage the data of a graph system as a record of type α *GS*. The type parameter α denotes the type of darts. We require that there is a linear order defined on α. As in our application α will always be *nat*, this is not a problem.

There are two kinds of record components: those members which describe the topology of the planar graph (fig. 4.2), and those members which represent the variables on darts (fig. 4.3); all variables have values in $\mathbb{R}$. Note that although e.g. the type of *gs*-σ is stated to be $\alpha \Rightarrow$ *real*, it really is α *GS* $\Rightarrow \alpha \Rightarrow$ *real*, taking an explicit graph system record as parameter.

What turns a structure *gs* of type α *GS* into a graph system is the set of axioms it has to fulfill. All of these axioms are stated relative to *gs*. Because we have many axioms, it would be nice to state them as concise as possible. The Isabelle *locale* mechanism [1] fits our purpose. It allows entering a context in which chosen free

gs-Face :: $\alpha \Rightarrow \alpha$ set
gs-Edge :: $\alpha \Rightarrow \alpha$ set
gs-Node :: $\alpha \Rightarrow \alpha$ set

gs-darts :: α set
gs-edge :: $\alpha \Rightarrow \alpha$
gs-face :: $\alpha \Rightarrow \alpha$
gs-node :: $\alpha \Rightarrow \alpha$

gs-edges :: α set
gs-nodes :: α set
gs-faces :: α set

gs-edgerep :: $\alpha \Rightarrow \alpha$
gs-noderep :: $\alpha \Rightarrow \alpha$
gs-facerep :: $\alpha \Rightarrow \alpha$

gs-adjacent :: $\alpha \Rightarrow \alpha \Rightarrow$ bool
gs-commonquad :: $\alpha \Rightarrow \alpha \Rightarrow$ bool
gs-separable :: $\alpha \Rightarrow$ bool
gs-separable′ :: α set $\Rightarrow \alpha \Rightarrow$ bool
gs-tri :: $\alpha \Rightarrow$ nat
gs-node-type :: $\alpha \Rightarrow$ nat $\Rightarrow$ bool

Figure 4.2: Planar Graph Record Components

gs-σ :: $\alpha \Rightarrow$ real
gs-τ :: $\alpha \Rightarrow$ real
gs-ye :: $\alpha \Rightarrow$ real
gs-yn :: $\alpha \Rightarrow$ real
gs-sol :: $\alpha \Rightarrow$ real
gs-azim :: $\alpha \Rightarrow$ real

gs-tau-sigma :: $\alpha \Rightarrow$ real
gs-sigma-qrtet :: $\alpha \Rightarrow$ real
gs-sigma32-qrtet :: $\alpha \Rightarrow$ real
gs-sigma1-qrtet :: $\alpha \Rightarrow$ real

gs-sigma-quad :: $\alpha \Rightarrow$ real
gs-tau-quad :: $\alpha \Rightarrow$ real
gs-squad-quad :: $\alpha \Rightarrow$ real
gs-sol-quad :: $\alpha \Rightarrow$ real
gs-azim1-quad :: $\alpha \Rightarrow$ real
gs-azim2-quad :: $\alpha \Rightarrow$ real

Figure 4.3: Real Variable Record Components

variables are *fixed*, that means within the context they are treated like constants, but are generalized on exiting the context. We choose *gs* as fixed variable and can then view all of the record components in Figures 4.2 and 4.3 as constants with a built-in implicit *gs* parameter. In the locale context we define e.g.

$$\sigma \quad = \quad gs\text{-}\sigma\ gs.$$

We introduce such an abbreviation for all of our record components and use these abbreviations from now on (except at those rare occasions where we look at a graph system from the outside).

4.2.1 Topology of a Graph System

Let us first make sense of the components shown in Figure 4.2. The leftmost box displays the components which model a planar graph in hypermap representation: the set of *darts*, and the *edge*, *node* and *face* permutations. We have no axioms actually enforcing that these ought to be permutations.

All other components displayed in Figure 4.2 are defined in terms of these 4 primitive components. The *Face*, *Edge* and *Node* functions assign to each dart its equivalence class. The definition of those functions listed in Figure 4.4 uses the notion of *Orbit*:

$$Orbit\ f\ s = \{f^n s \mid n \in \mathbb{N}\} = \{s, f\ s, f\ (f\ s), \ldots\} \tag{4.6}$$

Sometimes we need to represent such an equivalence class by a single dart unique

$$\begin{aligned} \mathit{Face} &= \mathit{Orbit\ face} \\ \mathit{Edge} &= \mathit{Orbit\ edge} \\ \mathit{Node} &= \mathit{Orbit\ node} \end{aligned}$$

Figure 4.4: Axioms for *Face*, *Edge* and *Node*

$$\forall\, \alpha.\ \mathit{edgerep}\ \alpha = \mathit{Min}\ (\mathit{Edge}\ \alpha)$$
$$\forall\, \alpha.\ \mathit{facerep}\ \alpha = \mathit{Min}\ (\mathit{Face}\ \alpha)$$
$$\forall\, \alpha.\ \mathit{noderep}\ \alpha = \mathit{Min}\ (\mathit{Node}\ \alpha)$$

Figure 4.5: Axioms for *edgerep*, *facerep* and *noderep*

to the class. This is why we require the type α of darts to be totally ordered: so that we can pick a single dart in a unique way out of each equivalence class. The *edgerep*, *noderep* and *facerep* functions assign to each dart the unique representing dart that belongs to the same equivalence class (fig. 4.5). We also want to be able to address *all* faces, or *all* edges, or *all* nodes. Thus, for each permutation, we form the set of all representatives of that permutation (fig. 4.6).

Finally, the rightmost box of Figure 4.2 is made up of further notions which enrich the topology related vocabulary of graph systems. Their definitions are listed in Figure 4.7.

4.2.2 3-Space Interpretation of a Graph System

To understand the real variable components of a graph system, it is helpful to recall how tame graphs originate from a packing of three-dimensional balls of radius 1 in 3-space. Given such a packing, pick any ball; this ball serves as the *origin*. Now select all balls the centers of which have a distance from the center of the origin of less than or equal to 2.51. Connect then all such centers with straight lines if their distance is less than $2\ t_0 = 2.51$. Project all of these lines onto the ball at the origin, and you have a spherical plane graph. After some normalization of this spherical plane graph you get a tame graph.

This means that a node of a tame graph corresponds to the center of a ball. And the edges of a tame graph correspond to the straight line connections between those centers. And the faces of a tame graph are basically a partitioning of the volume around the origin.

The real variable components refer to this 3-space interpretation of tame graphs. The real variable $\mathit{yn}\ \alpha$ interprets the dart α as a node; hence it must be invariant

$$\forall\, \alpha.\ \mathit{edges} = \{\ \alpha \in \mathit{darts}.\ \mathit{edgerep}\ \alpha = \alpha\ \}$$
$$\forall\, \alpha.\ \mathit{faces} = \{\ \alpha \in \mathit{darts}.\ \mathit{facerep}\ \alpha = \alpha\ \}$$
$$\forall\, \alpha.\ \mathit{nodes} = \{\ \alpha \in \mathit{darts}.\ \mathit{noderep}\ \alpha = \alpha\ \}$$

Figure 4.6: Axioms for *edges*, *faces* and *nodes*

$$\forall\,\alpha\,\beta.\ adjacent\ \alpha\ \beta = (\exists n \in Node\ \alpha.\ edge\ \alpha \in Node\ \beta)$$
$$\forall\,\alpha\,\beta.\ commonquad\ \alpha\ \beta = (\exists F \in Node\ \alpha.\ card\ (Face\ F) = 4\ and\ facerep\ F \in facerep\ `\ Node\ \beta)$$
$$\forall\,\alpha.\ separable\ \alpha = (card\ (Node\ \alpha) = 5\ and\ (\exists F \in Node\ \alpha.\ 5 \le card\ (Face\ F)))$$
$$\forall\,S\ \alpha.\ separable'\ S\ \alpha = (separable\ \alpha\ and\ (\forall \beta \in S.\ \neg\ (adjacent\ \alpha\ \beta\ or\ commonquad\ \alpha\ \beta)))$$
$$\forall\,\alpha.\ tri\ \alpha = card\ (Node\ \alpha \cap \{\ \beta\ .\ card\ (Face\ \beta) = 3\ \})$$
$$\forall\,\alpha\ n.\ node\text{-}type\ \alpha\ n = (card\ (Node\ \alpha) = n \wedge (\forall \beta \in Node\ \alpha.\ card\ (Face\ \beta) = 3))$$

Figure 4.7: Further Topology Axioms

under the *node* permutation. It denotes the distance that the corresponding center of a ball has to the origin. Hence we have $2 \le yn\ \alpha \le 2\ t_0$.

The real variable *ye* α interprets α as an edge. It is invariant under the *edge* permutation and measures the length of the straight line between two balls it corresponds to. Again, we have $2 \le ye\ \alpha \le 2\ t_0$.

The real variable *sol* α interprets α as a face. It measures the size of the area of the surface of the ball at the origin that this face corresponds to. It is invariant under the face permutation. Because the whole surface of a unit ball is $4\ \pi$, we have $0 \le sol\ \alpha \le 4\ \pi$.

To explain the real variable *azim* α, let us first interpret α as an arc on the surface of the ball at the origin. Applying the *node* permutation to α yields another dart α' which we also interpret as such an arc. The two arcs have a point p in common, the projection of the center of the ball that α corresponds to. Now *azim* α measures the spherical angle between those two arcs at p. Because *azim* α is an angle, we have $0 \le azim\ \alpha \le 2\ \pi$. We furthermore know the sum of all angles around a point:

$$\forall N \in nodes.\ 2 * \pi = \sum \alpha \in Node\ N.\ azim\ \alpha \tag{4.7}$$

In the plane, the sum of all inner angles of a triangle is π. For a spherical triangle, this is not true. Girard's Formula says that the difference between the sum of all inner spherical angles of a spherical triangle and π is just the area of that spherical triangle. Generalizing this result to faces with ≥ 3 edges gives:

$$\forall x \in darts.\ sol\ x = -\ real\ (card\ (Face\ x) - 2) * \pi + \sum \alpha \in Face\ x.\ azim\ \alpha \tag{4.8}$$

between the ends of that straight line and the origin.

The real variables $\sigma\ \alpha$ (*the score*) and $\tau\ \alpha$ are related to the density of the volume the face α corresponds to. They are connected via

$$\forall x \in darts.\ \tau\ x = sol\ x * \zeta * pt - \sigma\ x \tag{4.9}$$

where $pt = 4\ \arctan(\sqrt{2}/5) - \frac{\pi}{3}$ and $\zeta = 1/(2\ \arctan(\sqrt{2}/5))$. For the graph system to be *contravening*, that is to qualify as part of a packing with highest possible density,

$$8 * pt \le \sum \alpha \in faces.\ \sigma\ \alpha \tag{4.10}$$

must hold.

The other real variables in the middle and rightmost box in Figure 4.3 are variations on the real variables in the first box; they are specified only in certain situations. Their relationship with the real variables from the first box is given by the axioms in Figure 4.11. The significance of these other variables lies in the existence of axioms which have been converted from a database of inequalities. See Appendix A for the complete list of these axioms.

$\forall x \in darts.\ \sigma\ x = \sigma\ (face\ x)$
$\forall x \in darts.\ ye\ x = ye\ (edge\ x)$
$\forall x \in darts.\ yn\ x = yn\ (node\ x)$
$\forall x \in darts.\ sol\ x = sol\ (face\ x)$

Figure 4.8: Axioms for Invariance under Permutation

$\forall x \in darts.\ 2 \leq yn\ x$
$\forall x \in darts.\ 2 \leq ye\ x$
$\forall x \in darts.\ 0 \leq azim\ x$
$\forall x \in darts.\ 0 \leq sol\ x$
$\forall x \in darts.\ yn\ x \leq 2 * t_0$
$\forall x \in darts.\ ye\ x \leq 2 * t_0$
$\forall x \in darts.\ azim\ x \leq 2 * \pi$
$\forall x \in darts.\ sol\ x \leq 4 * \pi$

Figure 4.9: Axioms for Basic Geometrical Bounds

$\forall x \in darts.\ card\ (Face\ x) = 3$ **implies** $0 \leq \tau\ x$
$\forall x \in darts.\ card\ (Face\ x) = 4$ **implies** $1317\ /\ 10000 \leq \tau\ x$
$\forall x \in darts.\ card\ (Face\ x) = 5$ **implies** $27113\ /\ 100000 \leq \tau\ x$
$\forall x \in darts.\ card\ (Face\ x) = 6$ **implies** $41056\ /\ 100000 \leq \tau\ x$
$\forall x \in darts.\ card\ (Face\ x) = 7$ **implies** $54999\ /\ 100000 \leq \tau\ x$
$\forall x \in darts.\ card\ (Face\ x) = 8$ **implies** $6045\ /\ 10000 \leq \tau\ x$
$\forall x \in darts.\ card\ (Face\ x) = 3$ **implies** $\sigma\ x \leq pt$
$\forall x \in darts.\ card\ (Face\ x) = 4$ **implies** $\sigma\ x \leq 0$
$\forall x \in darts.\ card\ (Face\ x) = 5$ **implies** $\sigma\ x \leq -5704\ /\ 100000$
$\forall x \in darts.\ card\ (Face\ x) = 6$ **implies** $\sigma\ x \leq -11408\ /\ 100000$
$\forall x \in darts.\ card\ (Face\ x) = 7$ **implies** $\sigma\ x \leq -17112\ /\ 100000$
$\forall x \in darts.\ card\ (Face\ x) = 8$ **implies** $\sigma\ x \leq -22816\ /\ 100000$

Figure 4.10: Bounds for σ and τ from [16, Lemma 20.2]

$\forall x \in darts.\ card\ (Face\ x) = 3$ **implies**
$sigma32\text{-}qrtet\ x = sigma\text{-}qrtet\ x - 32\ /\ 10 * \zeta * pt * sol\ x$
$\forall x \in darts.\ card\ (Face\ x) = 3$ **implies**
$sigma1\text{-}qrtet\ x = sigma\text{-}qrtet\ x - \zeta * pt * sol\ x$
$\forall x \in darts.\ card\ (Face\ x) = 3$ **implies** $\sigma\ x = sigma\text{-}qrtet\ x \wedge \tau\ x = tau\text{-}sigma\ x$
$\forall x \in darts.\ card\ (Face\ x) = 4$ **implies** $\sigma\ x = sigma\text{-}quad\ x$
$\forall x \in darts.\ card\ (Face\ x) = 4$ **implies** $\tau\ x = tau\text{-}quad\ x$
$\forall x \in darts.\ card\ (Face\ x) = 4$ **implies** $squad\text{-}quad\ x = \sigma\ x \wedge sol\text{-}quad\ x = sol\ x \wedge$
$azim1\text{-}quad\ x = azim\ x \wedge azim2\text{-}quad\ x = azim\ (face\ x)$

Figure 4.11: Variations of Real Variables

$$\begin{aligned}
&\forall \alpha \in \mathit{nodes}.\ \mathit{node\text{-}type}\ \alpha\ 5\ \mathbf{implies}\\
&\quad (\forall \beta \in \mathit{nodes}.\ \alpha < \beta \wedge \mathit{node\text{-}type}\ \beta\ 5\ \mathbf{implies}\\
&\quad (\forall \gamma \in \mathit{nodes}.\ \beta < \gamma \wedge \mathit{node\text{-}type}\ \gamma\ 5\ \mathbf{implies}\\
&\quad (\forall \delta \in \mathit{nodes}.\ \gamma < \delta \wedge \mathit{node\text{-}type}\ \delta\ 5\ \mathbf{implies}\\
&\quad 55\ /\ 100 * 4 * pt \le (\textstyle\sum\ S \in \mathit{faces} \cap \mathit{facerep}\ `\ (\mathit{Node}\ \alpha \cup \mathit{Node}\ \beta \cup \mathit{Node}\ \gamma \cup \mathit{Node}\ \delta).\ \tau\ S)\\
&\quad \wedge (\textstyle\sum S \in \mathit{faces} \cap \mathit{facerep}\ `\ (\mathit{Node}\ \alpha \cup \mathit{Node}\ \beta \cup \mathit{Node}\ \gamma \cup \mathit{Node}\ \delta).\ \sigma\ S - pt) \le -\ 48\ /\ 100 * 4 * pt)))
\end{aligned}$$

$$\begin{aligned}
&\forall \alpha \in \mathit{nodes}.\ \mathit{node\text{-}type}\ \alpha\ 5\ \mathbf{implies}\\
&\quad (\forall \beta \in \mathit{nodes}.\ \alpha < \beta \wedge \mathit{node\text{-}type}\ \beta\ 5\ \mathbf{implies}\\
&\quad (\forall \gamma \in \mathit{nodes}.\ \beta < \gamma \wedge \mathit{node\text{-}type}\ \gamma\ 5\ \mathbf{implies}\\
&\quad 55\ /\ 100 * 3 * pt \le (\textstyle\sum\ S \in \mathit{faces} \cap \mathit{facerep}\ `\ (\mathit{Node}\ \alpha \cup \mathit{Node}\ \beta \cup \mathit{Node}\ \gamma).\ \tau\ S)\\
&\quad \wedge (\textstyle\sum S \in \mathit{faces} \cap \mathit{facerep}\ `\ (\mathit{Node}\ \alpha \cup \mathit{Node}\ \beta \cup \mathit{Node}\ \gamma).\ \sigma\ S - pt) \le -\ 48\ /\ 100 * 3 * pt))
\end{aligned}$$

$$\begin{aligned}
&\forall \alpha \in \mathit{nodes}.\ \mathit{node\text{-}type}\ \alpha\ 5\ \mathbf{implies}\\
&\quad (\forall \beta \in \mathit{nodes}.\ \alpha < \beta \wedge \mathit{node\text{-}type}\ \beta\ 5\ \mathbf{implies}\\
&\quad 55\ /\ 100 * 2 * pt \le (\textstyle\sum\ S \in \mathit{faces} \cap \mathit{facerep}\ `\ (\mathit{Node}\ \alpha \cup \mathit{Node}\ \beta).\ \tau\ S)\\
&\quad \wedge (\textstyle\sum S \in \mathit{faces} \cap \mathit{facerep}\ `\ (\mathit{Node}\ \alpha \cup \mathit{Node}\ \beta).\ \sigma\ S - pt) \le -\ 48\ /\ 100 * 2 * pt)
\end{aligned}$$

$$\begin{aligned}
&\forall \alpha \in \mathit{nodes}.\ \mathit{node\text{-}type}\ \alpha\ 5\ \mathbf{implies}\\
&\quad 55\ /\ 100 * 1 * pt \le (\textstyle\sum\ S \in \mathit{faces} \cap \mathit{facerep}\ `\ \mathit{Node}\ \alpha.\ \tau\ S)\\
&\quad \wedge (\textstyle\sum S \in \mathit{faces} \cap \mathit{facerep}\ `\ \mathit{Node}\ \alpha.\ \sigma\ S - pt) \le -\ 48\ /\ 100 * 1 * pt
\end{aligned}$$

Figure 4.12: Axioms from [16, Lemma 10.6]

$$\begin{aligned}
&\forall v_1 \in \mathit{nodes}.\ \mathit{separable}\ v_1\ \mathbf{implies}\\
&\quad \mathit{const\text{-}a}\ (\mathit{tri}\ v_1) \le (\textstyle\sum F \in \mathit{faces} \cap \mathit{facerep}\ `\ \mathit{Node}\ v_1.\ \tau\ F\ /\ pt - \mathit{const\text{-}d}\ (\mathit{card}\ (\mathit{Face}\ F)))
\end{aligned}$$

$$\begin{aligned}
&\forall v_1 \in \mathit{nodes}.\ \mathit{separable}\ v_1\ \mathbf{implies}\\
&\quad (\forall v_2 \in \mathit{nodes}.\ v_1 < v_2\ \mathbf{and}\ \mathit{separable}'\ \{v_1\}\ v_2\ \mathbf{implies}\\
&\quad \mathit{const\text{-}a}\ (\mathit{tri}\ v_1) + \mathit{const\text{-}a}\ (\mathit{tri}\ v_2)\\
&\quad \le (\textstyle\sum F \in \mathit{faces} \cap \mathit{facerep}\ `\ (\mathit{Node}\ v_1 \cup \mathit{Node}\ v_2).\ \tau\ F\ /\ pt - \mathit{const\text{-}d}\ (\mathit{card}\ (\mathit{Face}\ F))))
\end{aligned}$$

$$\begin{aligned}
&\forall v_1 \in \mathit{nodes}.\ \mathit{separable}\ v_1\ \mathbf{implies}\\
&\quad (\forall v_2 \in \mathit{nodes}.\ v_1 < v_2\ \mathbf{and}\ \mathit{separable}'\ \{v_1\}\ v_2\ \mathbf{implies}\\
&\quad (\forall v_3 \in \mathit{nodes}.\ v_2 < v_3\ \mathbf{and}\ \mathit{separable}'\ \{v_1, v_2\}\ v_3\ \mathbf{implies}\\
&\quad \mathit{const\text{-}a}\ (\mathit{tri}\ v_1) + \mathit{const\text{-}a}\ (\mathit{tri}\ v_2) + \mathit{const\text{-}a}\ (\mathit{tri}\ v_3)\\
&\quad \le (\textstyle\sum F \in \mathit{faces} \cap \mathit{facerep}\ `\ (\mathit{Node}\ v_1 \cup \mathit{Node}\ v_2 \cup \mathit{Node}\ v_3).\ \tau\ F\ /\ pt - \mathit{const\text{-}d}\ (\mathit{card}\ (\mathit{Face}\ F)))))
\end{aligned}$$

Figure 4.13: Axioms from [16, Lemma 22.12]

4.2.3 Additional Constraints of a Graph System

We are about to complete the specification of the axioms of a graph system. This subsection enumerates all axioms that are still missing.

First, this axiom is derived from [13, Group 4, rule 1]:

$$\forall \alpha \in \mathit{nodes}.\ \mathit{node\text{-}type}\ \alpha\ 4\ \mathbf{implies}\ (\forall \beta \in \mathit{Node}\ \alpha.\ \sigma\ \beta \le 33\ /\ 100 * pt) \tag{4.11}$$

Second, here is an axiom derived from [13, Group 4, rule 3]:

$$\begin{aligned}
&\forall \alpha \in \mathit{nodes}.\ \mathit{node\text{-}type}\ \alpha\ 5\ \mathbf{implies}\\
&\quad (\sum \beta \in \mathit{Node}\ \alpha.\ \sigma\ \beta + 419351\ /\ 1000000 * \mathit{sol}\ \beta - 2856354\ /\ 10000000) \le 0
\end{aligned} \tag{4.12}$$

Finally, Figures 4.12 and 4.13 complete the set of axioms.

Thus we have defined the predicate *GraphSystem* of type $\alpha\ GS \Rightarrow \mathit{bool}$ which is true for some *gs* if *gs* fulfills all the axioms mentioned in Section 4.2.

4.3 Generating and Running the Basic Linear Programs

Each graph system axiom involving real variables is a generator for a set of linear inequalities in these real variables. For each tame graph, we assume that it fulfills all graph system axioms. We then run the axioms and produce a system of linear inequalities. If we are successful in showing that this system is infeasible, we have shown that the given tame graph cannot be a graph system, and thus constitutes no counter example to the Kepler conjecture.

We first need a connection between graph system and tame graph:

definition *func-eq* :: α *GS* $\Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow$ *bool*
where
func-eq gs f g = ($\forall$ *d*. *d* $\in$ *gs-darts gs* $\longrightarrow$ *f d* = *g d*)

definition *PGS* :: *nat GS* $\Rightarrow$ (*nat* $\times$ *nat* $\times$ *nat*) *NatTreeMap* $\Rightarrow$ *bool*
where (4.13)
PGS gs S = (*GraphSystem gs*
$\wedge$ *gs-darts gs* = *dom* (*eval S*)
$\wedge$ *func-eq gs* (*gs-face gs*) (*map-face S*)
$\wedge$ *func-eq gs* (*gs-edge gs*) (*map-edge S*)
$\wedge$ *func-eq gs* (*gs-node gs*) (*map-node S*))

For a given tame graph S, say $S = graph\text{-}47$, we can then enter the context *PGS gs S*. The HOL Computing Library (HCL) allows us to perform computations within this context. If the HCL did not have that capability, it would have been impossible to apply the HCL to our problem, because all of our computations need to be done in a world which has the underlying implicit assumption *PGS gs S*. Our goal is to prove *False* in this world, thereby showing that this is a world which is not real.

How do we execute an axiom? First note that by linking graph system and tame graph, the permutation functions of the graph system become executable. We provide theorems to the HCL such that from the executability of the permutation functions the executability of all axioms follows.

Say the axiom has the form

$$\forall\, x \in B.\, P\, x$$

If B is a finite set such that $B = \{b_1, \ldots, b_n\}$, then executing this axiom means converting it into the form $P\ b_1 \wedge \ldots \wedge P\ b_n$. In our situation, B is often given as the orbit of a permutation function, for example $B = \mathit{Orbit\ face}\ d$. Then executing the axiom results in

$$P\ d \wedge P(\mathit{face}\ d) \wedge P(\mathit{face}(\mathit{face}\ d)) \wedge \ldots$$

where the conjunction is finite if the orbit is. The nice thing about finite orbits is therefore that they come with a built-in traversal strategy, i.e. we know how to visit every element of the set exactly once. If X is a set with such a traversal strategy, and Y is a set such that the membership test $y \in Y$ is executable (which is e.g. true if Y has a traversal strategy), then $X \cap Y$ has also a traversal strategy: just follow the traversal strategy of X, skipping elements if they are not in Y. Of course,then $X - Y$ also has a traversal strategy: follow the one of X, skipping elements if they are in Y. The case $X \cup Y$ is problematic, even if both X and Y have a traversal strategy. Our solution in this situation is to find a set C with traversal strategy such that $X \cup Y = C \cap (X \cup Y)$. Then $C \cap (X \cup Y)$ is clearly traversable because C has a traversal strategy and the membership test for $X \cup Y$ is executable. All of the sets in our application are subsets of *darts*. If *darts* is a traversable set, then this reformulation is always possible. Note that although the requirement to visit an element not more than once is not really necessary for executing $\forall$, it is nevertheless essential for executing $\sum$.

We showed in [28] how to define and reason about functions on orbits with the help of the *While* and *For* combinators. Using the techniques presented there we have defined a fold functional for orbits, and proven theorems about the relationship of that fold functional with the one for finite sets presented in [24]. This allowed us to reuse many results already available in Isabelle, for example when making the *Min* function executable on orbits, which is needed for making *facerep* etc. executable.

In many axioms we used instead of the normal implication operator $\longrightarrow$ the short-circuit operator **implies** (fig. 2.10). Note that this is essential in order to be able to execute certain axioms like those in Figure 4.12 or 4.13 in reasonable time.

The result of executing a graph specification is a large conjunction of equalities and inequalities. We perform a normalization step to turn this large conjunction into matrix form. We then approximate the matrix by the method mentioned in Section 3.7.4. After this, we calculate a priori bounds as explained in Section 3.8. Then the linear program is ripe for trying to prove its infeasibility. Applying the method of Section 3.5, we manage to prove *False* for about 92.5% of all tame graphs. For example, for the tame graph *graph*-47 we prove

$$PGS\ gs\ graph\text{-}47 \Longrightarrow False$$

Detailed results are listed in Appendix B. Let's do a quick sanity check. The number of the tame graph corresponding to the face-centered cubic packing is 901, the number of the one corresponding to the hexagonal-close packing is 880. Looking both numbers up shows that our methods failed for them, we could not prove $PGS\ gs\ graph\text{-}880 \Longrightarrow False$ or $PGS\ gs\ graph\text{-}901 \Longrightarrow False$. And that is how it should be.

Future work is to look at more complicated linear programs than the basic linear programs, and thereby to extend the methods presented in this thesis to tackle the remaining tame graphs.

APPENDIX **A**

Graph System Axioms from the Inequality Database

This appendix lists those graph system axioms which have been converted from the *database of inequalities* which can be retrieved as HOL-light specification from [11].

The axioms are either for triangular faces, or for quadrilateral faces. For triangular faces, they hold for a face F under the assumption *tetra-bound F*, for quadrilateral faces they hold under the assumption *quad-bound F*. These predicates are local to the graph system specification and not exported, their definition is:

$$\begin{aligned}
&\textit{tetra-bound } F \equiv \\
&\quad \textit{card } (\textit{Face } F) = 3 \\
&\quad \wedge 2 \leq \textit{yn } F \wedge \textit{yn } F \leq 251/100 \\
&\quad \wedge 2 \leq \textit{yn } (\textit{face } F) \wedge \textit{yn } (\textit{face } F) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{yn } (\textit{face } (\textit{face } F)) \wedge \textit{yn } (\textit{face } (\textit{face } F)) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{ye } (\textit{face } F) \wedge \textit{ye } (\textit{face } F) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{ye } (\textit{face } (\textit{face } F)) \wedge \textit{ye } (\textit{face } (\textit{face } F)) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{ye } F \wedge \textit{ye } F \leq 251/100
\end{aligned} \tag{A.1}$$

$$\begin{aligned}
&\textit{quad-bound } F \equiv \\
&\quad \textit{card } (\textit{Face } F) = 4 \\
&\quad \wedge 2 \leq \textit{yn } F \wedge \textit{yn } F \leq 251/100 \\
&\quad \wedge 2 \leq \textit{yn } (\textit{face } F) \wedge \textit{yn } (\textit{face } F) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{yn } (\textit{face } (\textit{face } F)) \wedge \textit{yn } (\textit{face } (\textit{face } F)) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{yn } (\textit{face } (\textit{face } (\textit{face } F))) \wedge \textit{yn } (\textit{face } (\textit{face } (\textit{face } F))) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{ye } (\textit{face } F) \wedge \textit{ye } (\textit{face } F) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{ye } (\textit{face } (\textit{face } F)) \wedge \textit{ye } (\textit{face } (\textit{face } F)) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{ye } (\textit{face } (\textit{face } (\textit{face } F))) \wedge \textit{ye } (\textit{face } (\textit{face } (\textit{face } F))) \leq 251/100 \\
&\quad \wedge 2 \leq \textit{ye } F \wedge \textit{ye } F \leq 251/100
\end{aligned} \tag{A.2}$$

Because in this thesis we look only at graph systems corresponding to basic linear programs, the definitions could actually be expressed in a simpler way:

$$\textit{tetra-bound } F \equiv \textit{card } (\textit{Face } F) = 3 \tag{A.3}$$

$$\textit{quad-bound } F \equiv \textit{card } (\textit{Face } F) = 4 \tag{A.4}$$

$J_{16189133}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -1369\ /\ 10000 + sigma32\text{-}qrtet\ x + 1966\ /\ 10000 * azim\ x < 0$
$J_{49987949}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 190249\ /\ 1000000 + sigma\text{-}qrtet\ x + -446634\ /\ 1000000 * sol\ x < 0$
$J_{53415898}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow sigma1\text{-}qrtet\ x \leq 0$
$J_{73203677}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -13225\ /\ 10000 + sigma\text{-}qrtet\ x + 419351\ /\ 1000000 * sol\ x + 64713719\ /\ 100000000 * azim\ x < 0$
$J_{98170671}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -2114190\ /\ 10000000 + sigma\text{-}qrtet\ x + 419351\ /\ 1000000 * sol\ x + -610397\ /\ 10000000 * azim\ x < 0$
$J_{106537269}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -208\ /\ 1000 + sigma1\text{-}qrtet\ x + 1689\ /\ 10000 * azim\ x < 0$
$J_{170403135}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 5974\ /\ 10000 + sigma32\text{-}qrtet\ x + -4233\ /\ 10000 * azim\ x < 0$
$J_{195296574}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 45\ /\ 10000 + sigma32\text{-}qrtet\ x + 953\ /\ 10000 * azim\ x < 0$
$J_{208809199}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 8638\ /\ 10000 - azim\ x < 0$
$J_{221945658}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 3683\ /\ 10000 + sigma1\text{-}qrtet\ x + -2993\ /\ 10000 * azim\ x < 0$
$J_{254627291}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -3442\ /\ 10000 + sigma1\text{-}qrtet\ x + 2529\ /\ 10000 * azim\ x < 0$
$J_{382430711}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 4582620\ /\ 10000000 + sol\ x + -320937\ /\ 1000000 * ye\ (face\ x) + -320937\ /\ 1000000 * ye\ (face\ (face\ x)) + -320937\ /\ 1000000 * ye\ x + 152679\ /\ 1000000 * yn\ x + 152679\ /\ 1000000 * yn\ (face\ x) + 152679\ /\ 1000000 * yn\ (face\ (face\ x)) < 0$
$J_{507227930}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 651760\ /\ 10000000 + azim\ x + 153598\ /\ 1000000 * yn\ (face\ x) + 153598\ /\ 1000000 * yn\ (face\ (face\ x)) + 153598\ /\ 1000000 * ye\ (face\ (face\ x)) + 153598\ /\ 1000000 * ye\ x + -498\ /\ 1000 * yn\ x + -76446\ /\ 100000 * ye\ (face\ x) < 0$
$J_{539256862}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 41110\ /\ 100000 + sigma\text{-}qrtet\ x + -37898\ /\ 100000 * azim\ x < 0$
$J_{544014470}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 16183310\ /\ 10000000 + 199235\ /\ 1000000 * ye\ (face\ x) + 199235\ /\ 1000000 * ye\ (face\ (face\ x)) + 199235\ /\ 1000000 * ye\ x + -377076\ /\ 1000000 * yn\ x + -377076\ /\ 1000000 * yn\ (face\ x) + -377076\ /\ 1000000 * yn\ (face\ (face\ x)) - sol\ x < 0$
$J_{568731327}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 27341020\ /\ 10000000 + -359894\ /\ 1000000 * yn\ (face\ x) + -359894\ /\ 1000000 * yn\ (face\ (face\ x)) + -359894\ /\ 1000000 * ye\ (face\ (face\ x)) + -359894\ /\ 1000000 * ye\ x + 3\ /\ 1000 * yn\ x + 685\ /\ 1000 * ye\ (face\ x) - azim\ x < 0$
$J_{584511898}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 5786316\ /\ 10000000 + sigma\text{-}qrtet\ x + 419351\ /\ 1000000 * sol\ x + -796456\ /\ 1000000 * azim\ x < 0$
$J_{586468779}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -\ pt + sigma\text{-}qrtet\ x \leq 0$
$J_{649712615}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -128213260\ /\ 100000000 + sigma1\text{-}qrtet\ x + 129119\ /\ 1000000 * ye\ (face\ x) + 129119\ /\ 1000000 * ye\ (face\ (face\ x)) + 129119\ /\ 1000000 * ye\ x + 845696\ /\ 10000000 * yn\ x + 845696\ /\ 10000000 * yn\ (face\ x) + 845696\ /\ 10000000 * yn\ (face\ (face\ x)) < 0$
$J_{710947756}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -1486650\ /\ 1000000 + sigma\text{-}qrtet\ x + 419351\ /\ 1000000 * sol\ x + 2\ /\ 10 * yn\ x + 2\ /\ 10 * yn\ (face\ x) + 2\ /\ 10 * yn\ (face\ (face\ x)) < 0$
$J_{776305271}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -5353\ /\ 10000 + sigma\text{-}qrtet\ x + 3302\ /\ 10000 * azim\ x < 0$
$J_{789045970}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -13582137\ /\ 10000000 + sigma\text{-}qrtet\ x + 10857\ /\ 100000 * yn\ x + 10857\ /\ 100000 * yn\ (face\ x) + 10857\ /\ 100000 * yn\ (face\ (face\ x)) + 10857\ /\ 100000 * ye\ (face\ x) + 10857\ /\ 100000 * ye\ (face\ (face\ x)) + 10857\ /\ 100000 * ye\ x < 0$
$J_{802409438}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 2550\ /\ 10000 + sigma32\text{-}qrtet\ x + -1083\ /\ 10000 * azim\ x < 0$
$J_{809197575}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -35641\ /\ 100000 + sigma\text{-}qrtet\ x + 419351\ /\ 1000000 * sol\ x + 499559\ /\ 10000000 * azim\ x < 0$
$J_{825495074}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -2866354\ /\ 10000000 + sigma\text{-}qrtet\ x + 419351\ /\ 1000000 * sol\ x < 0$
$J_{864218323}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -23021\ /\ 100000 + sigma\text{-}qrtet\ x + 142\ /\ 1000 * azim\ x < 0$
$J_{868828815}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -308526\ /\ 1000000 + sigma\text{-}qrtet\ x + 419351\ /\ 1000000 * sol\ x + 162028\ /\ 10000000 * azim\ x < 0$
$J_{927432550}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow 4666\ /\ 10000 + sigma1\text{-}qrtet\ x + -3897\ /\ 10000 * azim\ x < 0$
$J_{984463800}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -1874445\ /\ 1000000 + azim\ x < 0$
$J_{995444025}$	$\forall x \in darts.\ tetra\text{-}bound\ x \longrightarrow -287389\ /\ 1000000 + sigma\text{-}qrtet\ x + 37642101\ /\ 100000000 * sol\ x < 0$

$J_{15141595}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 6284\ /\ 10000 + squad\text{-}quad\ x + -3878\ /\ 10000 * azim1\text{-}quad\ x < 0$
$J_{18502666}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 166174\ /\ 100000 + squad\text{-}quad\ x + 419351\ /\ 1000000 * sol\text{-}quad\ x + -1582508\ /\ 1000000 * azim1\text{-}quad\ x < 0$
$J_{73283761}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 8341\ /\ 10000 + squad\text{-}quad\ x + -5301\ /\ 10000 * azim1\text{-}quad\ x < 0$
$J_{122375455}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 2955\ /\ 1000 + squad\text{-}quad\ x + -(pt * \zeta) * sol\text{-}quad\ x + -21406\ /\ 10000 * azim1\text{-}quad\ x < 0$
$J_{153920401}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 4893\ /\ 1000 + squad\text{-}quad\ x + -(32\ /\ 10 * pt * \zeta) * sol\text{-}quad\ x + -35294\ /\ 10000 * azim1\text{-}quad\ x < 0$
$J_{166451608}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -41717\ /\ 100000 + squad\text{-}quad\ x + 3\ /\ 10 * sol\text{-}quad\ x < 0$
$J_{277330628}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -3247\ /\ 1000 + azim1\text{-}quad\ x < 0$
$J_{310151857}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 57906\ /\ 10000 + squad\text{-}quad\ x + -456766\ /\ 100000 * azim1\text{-}quad\ x < 0$
$J_{322621318}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 9494\ /\ 1000 + squad\text{-}quad\ x + -30508\ /\ 10000 * azim1\text{-}quad\ x + -30508\ /\ 10000 * azim2\text{-}quad\ x < 0$
$J_{337637212}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 4126\ /\ 10000 + squad\text{-}quad\ x + -(32\ /\ 10 * pt * \zeta) * sol\text{-}quad\ x < 0$
$J_{393682353}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -3825\ /\ 10000 + squad\text{-}quad\ x + -(pt * \zeta) * sol\text{-}quad\ x + 2365\ /\ 10000 * azim1\text{-}quad\ x < 0$
$J_{396281725}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -15707\ /\ 10000 + squad\text{-}quad\ x + 5905\ /\ 10000 * azim1\text{-}quad\ x < 0$
$J_{408478278}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 6438\ /\ 10000 + squad\text{-}quad\ x + -(pt * \zeta) * sol\text{-}quad\ x + -316\ /\ 1000 * azim1\text{-}quad\ x < 0$
$J_{444643063}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 10472\ /\ 10000 + squad\text{-}quad\ x + -27605\ /\ 100000 * azim1\text{-}quad\ x + -27605\ /\ 100000 * azim2\text{-}quad\ x < 0$
$J_{465497818}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 5350181\ /\ 1000000 + squad\text{-}quad\ x + 419351\ /\ 1000000 * sol\text{-}quad\ x + -4611391\ /\ 1000000 * azim1\text{-}quad\ x < 0$
$J_{539320075}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 581446\ /\ 100000 + squad\text{-}quad\ x + -(pt * \zeta) * sol\text{-}quad\ x + -449461\ /\ 100000 * azim1\text{-}quad\ x < 0$
$J_{552698390}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -35926\ /\ 10000 + squad\text{-}quad\ x + 844\ /\ 1000 * azim1\text{-}quad\ x + 844\ /\ 1000 * azim2\text{-}quad\ x < 0$
$J_{574391221}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -4124\ /\ 10000 + squad\text{-}quad\ x + 1897\ /\ 10000 * azim1\text{-}quad\ x < 0$
$J_{616145964}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 577942\ /\ 100000 + squad\text{-}quad\ x + -(32\ /\ 10 * pt * \zeta) * sol\text{-}quad\ x + -425863\ /\ 100000 * azim1\text{-}quad\ x < 0$
$J_{655029773}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 20749\ /\ 10000 + squad\text{-}quad\ x + -15094\ /\ 10000 * azim1\text{-}quad\ x < 0$
$J_{657406669}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 1153\ /\ 1000 - azim1\text{-}quad\ x < 0$
$J_{676439533}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -895\ /\ 10000 + squad\text{-}quad\ x + 419351\ /\ 1000000 * sol\text{-}quad\ x + -342747\ /\ 1000000 * azim1\text{-}quad\ x < 0$
$J_{768057794}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -33\ /\ 100 + squad\text{-}quad\ x + -(32\ /\ 10 * pt * \zeta) * sol\text{-}quad\ x + 316\ /\ 1000 * azim1\text{-}quad\ x < 0$
$J_{775642319}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -1071\ /\ 1000 + squad\text{-}quad\ x + -(pt * \zeta) * sol\text{-}quad\ x + 4747\ /\ 10000 * azim1\text{-}quad\ x < 0$
$J_{974296985}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow -336909\ /\ 100000 + squad\text{-}quad\ x + 419351\ /\ 1000000 * sol\text{-}quad\ x + 974137\ /\ 1000000 * azim1\text{-}quad\ x < 0$
$J_{996268658}$	$\forall x \in darts.\ quad\text{-}bound\ x \longrightarrow 1317\ /\ 10000 + squad\text{-}quad\ x + -(pt * \zeta) * sol\text{-}quad\ x < 0$

APPENDIX **B**

Results of Running the Basic LPs

In this appendix we list our results of running our methods on the archive of tame graphs. For each tame graph, we assumed that it forms a graph system. By generating the corresponding basic linear program and trying to prove it infeasible we tried to show that this assumption was false. Our results are presented in tables of the following format:

#	Inconsistent	Time

The '#' column contains the number of the tame graph that has been examined. The numbering is chosen to correspond to the order of the tame graphs listed in [22]. A tame graph is in class n if all of its faces have at most n edges and there is at least one face with n edges. Class 3 ranges from #1 to #20, class 4 from #21 to #943, class 5 from #944 to #2488, class 6 from #2489 to #2726, class 7 from #2727 to #2749, and class 8 from #2750 to #2771.

The 'Inconsistent' column says 'Yes' if we have successfully shown the infeasibility of the basic linear program induced by the tame graph, and therefore shown the inconsistency of the corresponding graph system. If it says '?', we only know that our methods failed on this graph.

Finally, the 'Time' column tells us how many minutes the examination of the tame graph lasted. We used the SML mode of the HOL Computing Library. Each tame graph has been examined by its own Isabelle process. Each Isabelle process ran on a dedicated processor of a cluster of 32 four processor 2.4GHz Opteron 850 machines with 8 GB RAM per machine. The quickest process needed 8.4 minutes, the slowest 67. The examination of *all* tame graphs took about 7.5 hours of cluster runtime. This corresponds to about 40 days on a single processor machine.

We were able to prove the inconsistency of 2565 of the graph systems, and failed on 206. This yields a success rate of about 92.5%.

#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time
1	Yes	15.4	101	Yes	18.7	201	Yes	21.4	301	Yes	26.4	401	Yes	24.9
2	Yes	21.9	102	Yes	19.9	202	Yes	24.1	302	Yes	28.4	402	Yes	26.7
3	Yes	17.6	103	Yes	24.0	203	Yes	18.2	303	Yes	27.0	403	Yes	24.1
4	Yes	39.8	104	Yes	18.1	204	Yes	30.0	304	Yes	26.7	404	Yes	21.5
5	Yes	19.4	105	Yes	23.8	205	Yes	26.1	305	Yes	30.9	405	Yes	25.3
6	Yes	23.1	106	Yes	25.0	206	Yes	27.2	306	Yes	20.1	406	Yes	27.0
7	Yes	26.9	107	Yes	21.1	207	Yes	26.1	307	Yes	24.7	407	Yes	27.3
8	Yes	24.3	108	Yes	18.4	208	Yes	31.8	308	Yes	32.6	408	Yes	19.1
9	Yes	41.5	109	Yes	24.2	209	Yes	25.1	309	Yes	21.0	409	Yes	23.5
10	Yes	40.7	110	Yes	25.6	210	Yes	28.3	310	Yes	36.2	410	Yes	19.6
11	Yes	37.7	111	Yes	18.8	211	Yes	25.8	311	Yes	32.9	411	Yes	31.9
12	Yes	30.4	112	Yes	23.6	212	Yes	27.7	312	Yes	31.1	412	Yes	23.2
13	Yes	30.9	113	Yes	26.0	213	Yes	22.3	313	Yes	30.0	413	Yes	24.0
14	Yes	47.3	114	Yes	19.4	214	Yes	21.0	314	Yes	32.3	414	Yes	25.2
15	Yes	53.5	115	Yes	18.1	215	Yes	29.4	315	Yes	36.4	415	Yes	23.5
16	Yes	66.8	116	Yes	23.4	216	Yes	29.9	316	Yes	17.9	416	Yes	23.2
17	Yes	56.1	117	Yes	18.3	217	Yes	26.6	317	Yes	17.6	417	Yes	20.6
18	?	47.3	118	Yes	29.3	218	Yes	29.5	318	Yes	22.1	418	Yes	21.7
19	Yes	15.9	119	Yes	23.7	219	Yes	26.4	319	Yes	18.2	419	Yes	22.7
20	Yes	12.7	120	Yes	17.8	220	Yes	26.4	320	Yes	19.3	420	Yes	22.1
21	Yes	20.0	121	Yes	22.9	221	Yes	27.0	321	Yes	22.8	421	Yes	19.0
22	Yes	20.8	122	Yes	23.9	222	Yes	35.0	322	Yes	16.0	422	Yes	22.5
23	Yes	22.9	123	Yes	25.9	223	Yes	31.7	323	Yes	20.0	423	Yes	22.1
24	Yes	23.6	124	Yes	25.6	224	Yes	29.1	324	Yes	22.6	424	Yes	25.4
25	Yes	24.3	125	Yes	23.5	225	Yes	21.2	325	Yes	18.9	425	Yes	24.0
26	Yes	21.0	126	Yes	26.0	226	Yes	24.1	326	Yes	17.7	426	Yes	20.3
27	Yes	21.6	127	Yes	26.7	227	Yes	25.2	327	Yes	20.9	427	Yes	25.0
28	Yes	18.0	128	Yes	24.5	228	Yes	32.6	328	Yes	16.1	428	Yes	20.9
29	Yes	18.6	129	Yes	20.4	229	Yes	22.7	329	Yes	17.8	429	Yes	24.2
30	Yes	21.6	130	Yes	20.4	230	Yes	27.0	330	Yes	20.7	430	Yes	22.8
31	Yes	20.6	131	Yes	18.4	231	Yes	26.8	331	Yes	20.4	431	Yes	24.0
32	Yes	22.5	132	Yes	28.1	232	Yes	28.7	332	Yes	27.3	432	Yes	19.8
33	Yes	19.8	133	?	19.8	233	Yes	28.8	333	Yes	19.1	433	Yes	20.1
34	Yes	20.6	134	Yes	27.2	234	Yes	32.3	334	Yes	21.2	434	Yes	23.8
35	Yes	21.9	135	Yes	26.2	235	Yes	29.1	335	Yes	19.9	435	Yes	18.5
36	?	19.8	136	Yes	21.3	236	Yes	28.6	336	Yes	18.0	436	Yes	24.9
37	Yes	21.6	137	Yes	24.7	237	Yes	26.7	337	Yes	18.7	437	Yes	25.6
38	Yes	21.6	138	?	20.6	238	Yes	31.1	338	Yes	19.7	438	Yes	23.6
39	Yes	23.9	139	Yes	19.3	239	Yes	30.0	339	Yes	18.3	439	Yes	20.8
40	Yes	22.9	140	?	19.7	240	Yes	30.8	340	Yes	18.8	440	Yes	19.1
41	Yes	19.2	141	Yes	22.8	241	Yes	35.9	341	Yes	21.3	441	Yes	21.4
42	Yes	25.8	142	Yes	27.7	242	Yes	21.8	342	Yes	18.2	442	Yes	18.8
43	Yes	22.7	143	?	18.5	243	Yes	30.4	343	Yes	17.6	443	Yes	20.2
44	Yes	23.0	144	Yes	22.4	244	Yes	17.6	344	Yes	17.8	444	Yes	18.7
45	Yes	19.7	145	?	21.0	245	Yes	23.1	345	Yes	21.5	445	Yes	19.8
46	Yes	27.4	146	?	19.6	246	Yes	28.1	346	Yes	18.7	446	Yes	19.7
47	Yes	18.2	147	Yes	31.5	247	Yes	27.5	347	Yes	18.8	447	Yes	24.7
48	Yes	21.3	148	Yes	17.7	248	Yes	31.7	348	Yes	20.3	448	Yes	24.2
49	Yes	22.4	149	Yes	18.7	249	Yes	27.2	349	Yes	25.6	449	Yes	27.3
50	Yes	22.0	150	Yes	21.7	250	Yes	30.5	350	Yes	27.3	450	Yes	26.9
51	Yes	20.9	151	Yes	21.7	251	Yes	24.3	351	Yes	22.6	451	Yes	24.2
52	Yes	18.2	152	Yes	26.0	252	Yes	21.3	352	Yes	21.5	452	Yes	23.0
53	Yes	18.8	153	Yes	28.2	253	Yes	18.9	353	Yes	25.0	453	Yes	26.1
54	Yes	20.0	154	Yes	21.2	254	Yes	22.4	354	Yes	25.2	454	Yes	20.3
55	Yes	20.3	155	Yes	24.6	255	Yes	18.2	355	Yes	28.4	455	Yes	21.2
56	Yes	20.9	156	Yes	23.2	256	?	22.8	356	Yes	20.0	456	Yes	27.5
57	Yes	18.5	157	Yes	23.6	257	Yes	17.8	357	Yes	19.5	457	Yes	25.3
58	Yes	19.9	158	?	20.1	258	Yes	19.0	358	Yes	18.8	458	Yes	25.0
59	Yes	18.0	159	Yes	29.4	259	Yes	26.9	359	Yes	23.8	459	Yes	23.6
60	Yes	17.3	160	Yes	19.8	260	Yes	18.9	360	Yes	16.8	460	Yes	23.3
61	?	19.4	161	Yes	17.8	261	Yes	24.4	361	Yes	17.8	461	Yes	27.2
62	?	19.2	162	Yes	21.2	262	Yes	26.4	362	Yes	18.7	462	Yes	25.1
63	Yes	19.7	163	Yes	19.9	263	Yes	21.7	363	Yes	17.3	463	Yes	20.8
64	Yes	23.4	164	Yes	26.8	264	Yes	26.9	364	Yes	19.9	464	Yes	29.2
65	Yes	19.4	165	?	28.0	265	Yes	29.1	365	Yes	19.1	465	Yes	27.6
66	Yes	23.4	166	Yes	25.2	266	Yes	25.5	366	Yes	19.3	466	Yes	35.8
67	Yes	22.8	167	Yes	25.2	267	Yes	24.0	367	Yes	16.1	467	Yes	23.8
68	Yes	19.7	168	Yes	28.3	268	Yes	23.9	368	Yes	19.4	468	Yes	19.9
69	Yes	23.5	169	Yes	27.4	269	Yes	22.8	369	?	24.5	469	Yes	17.9
70	Yes	24.0	170	Yes	27.9	270	Yes	17.6	370	Yes	18.3	470	Yes	25.2
71	Yes	24.7	171	Yes	17.9	271	Yes	27.0	371	Yes	18.2	471	Yes	28.3
72	Yes	19.8	172	Yes	32.4	272	Yes	22.2	372	Yes	19.1	472	Yes	25.7
73	Yes	21.6	173	Yes	17.8	273	Yes	19.5	373	Yes	19.7	473	Yes	24.6
74	Yes	25.9	174	Yes	18.0	274	Yes	22.9	374	Yes	18.0	474	Yes	27.3
75	Yes	27.1	175	?	22.0	275	Yes	25.6	375	Yes	21.6	475	Yes	24.2
76	Yes	17.6	176	Yes	25.6	276	Yes	26.6	376	Yes	18.2	476	Yes	25.6
77	Yes	28.7	177	Yes	22.9	277	Yes	25.4	377	Yes	19.8	477	Yes	25.1
78	?	26.1	178	Yes	25.8	278	Yes	27.8	378	Yes	19.4	478	Yes	24.5
79	Yes	23.3	179	Yes	17.8	279	Yes	27.8	379	Yes	20.3	479	Yes	19.1
80	Yes	18.3	180	Yes	22.2	280	Yes	25.3	380	Yes	20.9	480	Yes	19.0
81	?	28.3	181	Yes	24.6	281	Yes	27.2	381	Yes	23.5	481	Yes	23.0
82	Yes	22.2	182	Yes	28.5	282	Yes	28.5	382	Yes	20.5	482	Yes	19.5
83	Yes	25.4	183	Yes	20.4	283	Yes	23.5	383	Yes	22.8	483	Yes	18.3
84	Yes	18.8	184	Yes	21.9	284	Yes	25.4	384	Yes	18.7	484	Yes	15.1
85	Yes	25.4	185	Yes	23.1	285	Yes	27.2	385	Yes	31.9	485	Yes	15.2
86	Yes	26.0	186	Yes	25.8	286	Yes	28.1	386	Yes	22.8	486	Yes	16.8
87	Yes	21.9	187	Yes	30.3	287	Yes	30.4	387	Yes	25.5	487	Yes	18.7
88	Yes	25.0	188	Yes	28.4	288	Yes	24.8	388	Yes	21.2	488	Yes	16.6
89	?	26.9	189	Yes	27.0	289	Yes	22.7	389	Yes	19.2	489	Yes	15.4
90	?	27.5	190	Yes	18.3	290	Yes	25.9	390	Yes	25.6	490	Yes	16.1
91	?	19.4	191	Yes	25.9	291	Yes	28.5	391	Yes	26.5	491	Yes	17.2
92	Yes	23.5	192	Yes	20.4	292	Yes	30.3	392	Yes	25.1	492	Yes	16.9
93	Yes	26.0	193	Yes	24.7	293	Yes	22.7	393	Yes	21.0	493	Yes	16.7
94	Yes	25.3	194	Yes	30.7	294	Yes	24.9	394	Yes	25.2	494	Yes	14.1
95	Yes	40.4	195	Yes	27.6	295	Yes	30.1	395	Yes	23.4	495	Yes	14.2
96	Yes	25.1	196	Yes	25.6	296	Yes	23.5	396	Yes	18.8	496	Yes	18.0
97	Yes	22.6	197	?	23.6	297	Yes	23.7	397	Yes	24.9	497	Yes	17.8
98	Yes	18.7	198	Yes	20.5	298	Yes	22.7	398	Yes	25.3	498	Yes	15.9
99	Yes	22.2	199	Yes	19.8	299	Yes	28.0	399	Yes	24.1	499	Yes	18.2
100	Yes	18.0	200	Yes	20.8	300	Yes	28.7	400	Yes	24.2	500	Yes	19.1

#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time
501	Yes	18.0	601	Yes	19.8	701	Yes	9.4	801	Yes	25.6	901	?	21.7
502	Yes	18.6	602	Yes	16.8	702	Yes	12.3	802	Yes	11.5	902	Yes	14.8
503	Yes	17.1	603	Yes	16.8	703	Yes	15.7	803	Yes	14.3	903	Yes	31.1
504	Yes	18.3	604	Yes	20.5	704	Yes	11.1	804	Yes	12.8	904	Yes	28.8
505	Yes	18.7	605	Yes	19.3	705	Yes	12.0	805	Yes	11.7	905	Yes	10.2
506	Yes	14.1	606	Yes	32.1	706	Yes	10.9	806	Yes	15.7	906	Yes	17.5
507	Yes	16.3	607	Yes	17.7	707	Yes	11.8	807	Yes	20.1	907	Yes	12.4
508	Yes	17.5	608	Yes	15.4	708	Yes	15.1	808	Yes	21.0	908	Yes	16.6
509	Yes	16.2	609	Yes	18.4	709	Yes	9.9	809	Yes	13.5	909	Yes	31.6
510	Yes	16.2	610	Yes	14.1	710	Yes	12.6	810	Yes	17.6	910	Yes	11.8
511	Yes	18.5	611	Yes	16.8	711	Yes	15.2	811	Yes	13.0	911	Yes	11.6
512	Yes	17.4	612	Yes	13.4	712	Yes	12.2	812	Yes	17.3	912	Yes	16.8
513	Yes	16.6	613	Yes	13.9	713	Yes	12.1	813	Yes	16.3	913	Yes	17.6
514	Yes	16.8	614	Yes	18.4	714	Yes	16.4	814	Yes	16.5	914	Yes	33.5
515	Yes	18.0	615	Yes	16.7	715	Yes	12.4	815	Yes	34.7	915	Yes	33.5
516	Yes	19.1	616	Yes	19.0	716	Yes	14.2	816	Yes	23.1	916	Yes	31.8
517	Yes	13.4	617	Yes	20.7	717	Yes	14.8	817	Yes	34.2	917	Yes	39.4
518	Yes	17.1	618	Yes	17.7	718	Yes	16.4	818	Yes	11.3	918	Yes	44.5
519	Yes	16.4	619	Yes	19.6	719	Yes	13.9	819	Yes	15.3	919	Yes	11.6
520	Yes	16.8	620	Yes	15.8	720	Yes	16.3	820	Yes	18.4	920	Yes	8.6
521	Yes	14.5	621	Yes	19.2	721	Yes	11.0	821	Yes	19.0	921	Yes	14.5
522	Yes	14.5	622	Yes	17.7	722	Yes	11.9	822	Yes	18.0	922	Yes	18.2
523	Yes	17.4	623	Yes	18.9	723	Yes	15.4	823	Yes	20.5	923	Yes	14.5
524	Yes	13.1	624	Yes	36.1	724	Yes	13.2	824	Yes	18.0	924	Yes	19.1
525	Yes	17.9	625	Yes	38.8	725	Yes	11.9	825	Yes	18.6	925	Yes	33.6
526	Yes	18.7	626	Yes	29.4	726	Yes	13.0	826	Yes	14.5	926	Yes	32.0
527	Yes	18.3	627	Yes	33.7	727	Yes	16.5	827	Yes	17.3	927	Yes	37.8
528	Yes	18.7	628	Yes	36.7	728	Yes	15.5	828	Yes	22.9	928	Yes	40.3
529	Yes	17.9	629	Yes	30.8	729	Yes	20.8	829	Yes	23.6	929	Yes	15.2
530	Yes	17.8	630	Yes	34.5	730	Yes	8.4	830	Yes	18.9	930	Yes	34.7
531	Yes	17.3	631	Yes	30.5	731	Yes	13.3	831	Yes	21.9	931	Yes	8.6
532	Yes	18.8	632	Yes	11.8	732	Yes	9.6	832	Yes	37.6	932	Yes	13.0
533	Yes	20.4	633	Yes	13.2	733	Yes	13.8	833	Yes	35.7	933	Yes	11.4
534	Yes	20.0	634	Yes	13.0	734	Yes	19.9	834	Yes	16.7	934	?	20.3
535	Yes	22.7	635	Yes	12.8	735	Yes	12.7	835	Yes	20.7	935	Yes	22.5
536	Yes	18.6	636	Yes	11.3	736	Yes	12.7	836	Yes	40.6	936	Yes	42.3
537	Yes	22.3	637	Yes	13.4	737	Yes	15.5	837	Yes	37.4	937	Yes	51.1
538	Yes	19.1	638	Yes	15.1	738	Yes	10.0	838	Yes	18.1	938	Yes	16.5
539	Yes	23.8	639	Yes	14.4	739	Yes	14.0	839	Yes	35.1	939	Yes	13.9
540	Yes	18.6	640	Yes	18.3	740	Yes	17.1	840	Yes	46.4	940	Yes	14.4
541	Yes	21.7	641	Yes	14.3	741	Yes	20.0	841	Yes	43.6	941	Yes	14.5
542	Yes	22.2	642	Yes	15.8	742	Yes	11.3	842	Yes	40.0	942	Yes	19.2
543	Yes	22.2	643	Yes	15.3	743	Yes	12.0	843	Yes	33.6	943	Yes	11.4
544	Yes	24.7	644	Yes	16.6	744	Yes	13.6	844	Yes	34.8	944	?	12.8
545	Yes	17.4	645	Yes	14.5	745	Yes	10.2	845	Yes	40.2	945	Yes	12.7
546	Yes	20.0	646	Yes	18.6	746	Yes	16.0	846	Yes	45.8	946	?	19.7
547	Yes	22.7	647	Yes	18.2	747	Yes	11.5	847	Yes	40.0	947	Yes	13.6
548	Yes	23.2	648	Yes	18.4	748	Yes	15.7	848	Yes	41.3	948	Yes	19.9
549	Yes	23.0	649	Yes	14.5	749	Yes	15.5	849	Yes	42.9	949	Yes	14.4
550	Yes	18.7	650	Yes	17.4	750	Yes	13.4	850	Yes	45.2	950	?	20.1
551	Yes	17.5	651	Yes	16.4	751	Yes	17.4	851	Yes	34.8	951	Yes	17.3
552	Yes	21.6	652	Yes	14.0	752	Yes	20.4	852	Yes	40.7	952	?	20.0
553	Yes	19.1	653	Yes	15.5	753	Yes	14.1	853	Yes	42.1	953	Yes	16.0
554	Yes	18.6	654	Yes	13.7	754	Yes	12.9	854	Yes	31.9	954	Yes	15.8
555	Yes	18.9	655	Yes	17.2	755	Yes	19.9	855	Yes	42.8	955	Yes	22.7
556	Yes	19.1	656	Yes	16.8	756	Yes	12.0	856	Yes	36.0	956	Yes	13.8
557	Yes	21.4	657	Yes	13.2	757	Yes	13.7	857	Yes	44.0	957	Yes	15.7
558	Yes	22.4	658	Yes	16.6	758	Yes	18.2	858	Yes	33.7	958	Yes	26.8
559	Yes	12.9	659	Yes	19.4	759	Yes	17.3	859	Yes	35.0	959	Yes	20.3
560	Yes	16.2	660	Yes	11.2	760	Yes	15.4	860	Yes	36.6	960	Yes	23.4
561	Yes	12.1	661	Yes	15.9	761	Yes	16.7	861	Yes	35.7	961	Yes	12.0
562	Yes	14.5	662	Yes	16.5	762	Yes	16.0	862	Yes	39.4	962	?	17.7
563	Yes	14.9	663	Yes	14.2	763	Yes	16.0	863	Yes	18.3	963	Yes	16.2
564	Yes	11.0	664	Yes	17.3	764	Yes	20.1	864	Yes	16.2	964	Yes	16.2
565	Yes	15.0	665	Yes	18.9	765	Yes	28.0	865	Yes	18.4	965	Yes	18.3
566	Yes	15.4	666	Yes	18.8	766	Yes	11.1	866	Yes	19.2	966	?	20.9
567	Yes	17.3	667	Yes	12.6	767	Yes	12.5	867	Yes	24.8	967	Yes	19.8
568	Yes	14.9	668	Yes	16.3	768	Yes	15.2	868	Yes	39.9	968	Yes	24.4
569	Yes	11.8	669	Yes	14.9	769	Yes	15.7	869	Yes	41.9	969	Yes	20.9
570	Yes	11.9	670	Yes	16.1	770	Yes	18.5	870	Yes	33.1	970	Yes	17.5
571	Yes	14.8	671	Yes	15.6	771	Yes	19.0	871	Yes	33.7	971	Yes	27.1
572	Yes	18.0	672	Yes	14.4	772	Yes	15.0	872	Yes	39.4	972	Yes	14.1
573	Yes	13.7	673	Yes	16.4	773	Yes	18.3	873	Yes	47.7	973	Yes	14.1
574	Yes	16.4	674	Yes	19.5	774	Yes	13.2	874	Yes	42.2	974	Yes	20.8
575	Yes	15.4	675	Yes	17.3	775	Yes	17.2	875	Yes	41.5	975	Yes	18.4
576	Yes	18.0	676	Yes	34.2	776	Yes	21.5	876	Yes	45.3	976	Yes	13.6
577	Yes	11.8	677	Yes	19.1	777	Yes	17.3	877	Yes	17.9	977	Yes	20.1
578	Yes	14.6	678	Yes	31.9	778	Yes	23.7	878	Yes	44.4	978	Yes	25.7
579	Yes	10.5	679	Yes	34.5	779	Yes	16.5	879	Yes	43.9	979	Yes	18.5
580	Yes	12.4	680	Yes	16.3	780	Yes	18.1	880	?	20.9	980	Yes	23.5
581	Yes	14.4	681	Yes	19.2	781	Yes	14.3	881	Yes	14.0	981	Yes	21.1
582	Yes	13.6	682	Yes	33.6	782	Yes	16.0	882	Yes	12.4	982	Yes	14.5
583	Yes	11.7	683	Yes	18.3	783	Yes	26.0	883	Yes	11.8	983	Yes	13.5
584	Yes	14.5	684	Yes	21.5	784	Yes	17.4	884	Yes	13.4	984	Yes	17.8
585	Yes	13.7	685	Yes	31.5	785	Yes	15.4	885	Yes	17.0	985	Yes	26.5
586	Yes	14.1	686	Yes	32.0	786	Yes	12.8	886	Yes	29.4	986	Yes	20.5
587	Yes	15.0	687	Yes	34.0	787	Yes	19.0	887	Yes	15.2	987	Yes	26.6
588	Yes	13.7	688	Yes	35.1	788	Yes	18.1	888	Yes	15.9	988	Yes	17.1
589	Yes	15.8	689	Yes	37.8	789	Yes	16.4	889	Yes	26.7	989	Yes	20.5
590	Yes	15.0	690	Yes	38.7	790	Yes	15.9	890	Yes	31.1	990	Yes	12.7
591	Yes	19.5	691	Yes	34.3	791	Yes	25.8	891	Yes	12.6	991	Yes	15.8
592	Yes	17.3	692	Yes	30.9	792	Yes	16.9	892	Yes	14.9	992	Yes	24.0
593	Yes	13.2	693	Yes	30.4	793	Yes	17.7	893	Yes	27.1	993	Yes	20.1
594	Yes	13.9	694	Yes	37.2	794	Yes	18.1	894	Yes	27.0	994	Yes	15.7
595	Yes	15.3	695	Yes	29.7	795	Yes	16.4	895	Yes	15.1	995	Yes	21.7
596	Yes	17.4	696	Yes	49.5	796	Yes	18.6	896	?	17.8	996	Yes	32.5
597	Yes	17.2	697	Yes	25.8	797	Yes	18.7	897	Yes	28.1	997	Yes	16.7
598	Yes	14.7	698	Yes	10.8	798	Yes	27.7	898	Yes	14.4	998	Yes	17.5
599	Yes	14.8	699	Yes	14.4	799	Yes	26.5	899	Yes	34.6	999	Yes	26.5
600	Yes	17.1	700	Yes	10.8	800	Yes	30.0	900	Yes	35.5	1000	Yes	26.9

#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time
1001	Yes	26.4	1101	Yes	19.2	1201	Yes	18.4	1301	Yes	15.0	1401	Yes	19.4
1002	Yes	17.9	1102	Yes	30.7	1202	Yes	17.2	1302	Yes	16.0	1402	Yes	13.8
1003	Yes	13.8	1103	Yes	16.8	1203	Yes	21.6	1303	Yes	20.8	1403	Yes	16.7
1004	Yes	17.1	1104	Yes	20.0	1204	Yes	21.0	1304	Yes	24.9	1404	Yes	14.6
1005	Yes	25.3	1105	Yes	18.6	1205	Yes	19.4	1305	?	14.0	1405	?	12.4
1006	Yes	20.5	1106	Yes	28.6	1206	Yes	18.9	1306	Yes	20.3	1406	?	13.0
1007	Yes	28.7	1107	Yes	23.5	1207	Yes	16.1	1307	Yes	20.8	1407	Yes	11.6
1008	Yes	11.5	1108	Yes	27.9	1208	Yes	20.4	1308	Yes	18.7	1408	?	15.4
1009	Yes	12.8	1109	Yes	15.8	1209	?	18.3	1309	Yes	18.6	1409	?	11.7
1010	Yes	16.2	1110	Yes	11.5	1210	Yes	20.7	1310	Yes	18.1	1410	?	14.3
1011	Yes	20.7	1111	Yes	13.3	1211	Yes	18.4	1311	Yes	16.9	1411	?	19.2
1012	Yes	17.1	1112	Yes	15.6	1212	Yes	23.7	1312	Yes	18.3	1412	Yes	13.7
1013	Yes	20.5	1113	Yes	17.6	1213	Yes	23.6	1313	Yes	17.5	1413	?	18.6
1014	Yes	21.1	1114	Yes	19.1	1214	?	19.0	1314	?	22.1	1414	?	14.7
1015	Yes	26.8	1115	Yes	19.5	1215	Yes	18.5	1315	Yes	24.3	1415	?	13.9
1016	Yes	24.9	1116	Yes	17.3	1216	Yes	18.3	1316	Yes	23.7	1416	Yes	22.5
1017	Yes	16.9	1117	Yes	22.2	1217	Yes	22.7	1317	Yes	20.4	1417	?	19.2
1018	Yes	20.5	1118	Yes	24.2	1218	?	19.8	1318	Yes	16.4	1418	?	14.7
1019	Yes	28.9	1119	Yes	25.7	1219	?	14.0	1319	Yes	20.5	1419	Yes	17.3
1020	Yes	16.0	1120	Yes	21.2	1220	Yes	20.5	1320	Yes	20.5	1420	Yes	14.7
1021	Yes	18.4	1121	Yes	20.4	1221	Yes	18.0	1321	Yes	21.2	1421	Yes	17.2
1022	Yes	14.1	1122	Yes	23.2	1222	?	21.4	1322	Yes	26.8	1422	Yes	13.8
1023	Yes	21.0	1123	Yes	25.7	1223	Yes	22.0	1323	Yes	18.3	1423	Yes	15.1
1024	Yes	24.3	1124	Yes	27.8	1224	Yes	20.9	1324	Yes	19.1	1424	Yes	12.8
1025	Yes	32.5	1125	Yes	44.4	1225	Yes	19.2	1325	Yes	22.2	1425	Yes	16.2
1026	Yes	16.3	1126	?	17.4	1226	Yes	23.9	1326	Yes	21.0	1426	Yes	17.9
1027	Yes	25.9	1127	Yes	19.3	1227	Yes	19.1	1327	Yes	24.3	1427	Yes	18.3
1028	Yes	21.2	1128	Yes	24.9	1228	Yes	20.6	1328	Yes	23.8	1428	Yes	13.4
1029	Yes	30.9	1129	Yes	14.3	1229	Yes	23.0	1329	Yes	20.9	1429	Yes	16.3
1030	Yes	24.7	1130	?	11.2	1230	Yes	23.8	1330	Yes	22.0	1430	Yes	15.5
1031	Yes	17.0	1131	?	14.3	1231	Yes	26.7	1331	Yes	26.0	1431	Yes	14.2
1032	Yes	24.4	1132	Yes	17.3	1232	Yes	29.5	1332	Yes	25.6	1432	Yes	15.9
1033	Yes	22.5	1133	Yes	19.9	1233	Yes	23.4	1333	Yes	19.6	1433	Yes	16.5
1034	Yes	22.7	1134	Yes	18.5	1234	Yes	20.4	1334	Yes	22.4	1434	Yes	14.2
1035	Yes	34.3	1135	Yes	16.0	1235	Yes	21.7	1335	Yes	25.1	1435	Yes	14.2
1036	Yes	19.9	1136	Yes	19.5	1236	Yes	33.9	1336	Yes	22.9	1436	?	13.9
1037	Yes	21.8	1137	?	18.3	1237	Yes	22.1	1337	Yes	21.2	1437	?	13.8
1038	Yes	17.5	1138	Yes	22.9	1238	Yes	24.9	1338	Yes	26.3	1438	Yes	23.8
1039	Yes	15.7	1139	Yes	18.0	1239	Yes	26.0	1339	Yes	24.1	1439	Yes	14.0
1040	Yes	20.3	1140	Yes	25.6	1240	Yes	29.2	1340	Yes	20.7	1440	?	15.3
1041	Yes	21.1	1141	Yes	18.8	1241	Yes	32.4	1341	Yes	21.8	1441	Yes	21.0
1042	Yes	26.7	1142	Yes	19.0	1242	Yes	11.7	1342	Yes	24.7	1442	Yes	16.8
1043	Yes	23.0	1143	Yes	25.8	1243	Yes	16.7	1343	Yes	25.4	1443	Yes	23.5
1044	Yes	24.8	1144	?	14.4	1244	Yes	11.5	1344	Yes	25.9	1444	Yes	13.9
1045	Yes	14.1	1145	Yes	13.8	1245	Yes	14.1	1345	Yes	26.6	1445	Yes	12.5
1046	Yes	17.8	1146	Yes	11.8	1246	Yes	16.9	1346	Yes	26.9	1446	Yes	13.7
1047	Yes	14.7	1147	Yes	15.6	1247	?	12.0	1347	Yes	34.7	1447	Yes	16.3
1048	Yes	21.9	1148	Yes	14.1	1248	?	15.1	1348	Yes	45.6	1448	Yes	12.5
1049	Yes	20.2	1149	Yes	13.7	1249	?	14.9	1349	Yes	44.3	1449	Yes	16.8
1050	Yes	18.4	1150	Yes	16.2	1250	Yes	15.9	1350	?	15.3	1450	Yes	14.8
1051	Yes	23.5	1151	Yes	18.3	1251	?	13.8	1351	Yes	17.8	1451	Yes	13.7
1052	Yes	16.9	1152	Yes	19.7	1252	Yes	14.6	1352	Yes	19.3	1452	Yes	19.6
1053	Yes	12.6	1153	Yes	24.0	1253	Yes	16.3	1353	Yes	19.0	1453	Yes	24.1
1054	Yes	26.5	1154	Yes	17.1	1254	?	19.1	1354	Yes	24.3	1454	Yes	20.4
1055	Yes	13.7	1155	Yes	18.3	1255	?	18.0	1355	Yes	21.7	1455	Yes	16.7
1056	Yes	13.5	1156	Yes	20.3	1256	Yes	19.4	1356	Yes	21.5	1456	Yes	19.8
1057	?	16.9	1157	Yes	17.5	1257	Yes	15.6	1357	Yes	21.3	1457	Yes	18.5
1058	Yes	20.4	1158	Yes	11.2	1258	Yes	17.0	1358	Yes	25.0	1458	Yes	12.1
1059	Yes	18.1	1159	Yes	14.1	1259	Yes	15.4	1359	Yes	21.5	1459	Yes	15.0
1060	Yes	14.0	1160	Yes	15.5	1260	Yes	18.3	1360	Yes	20.8	1460	Yes	15.3
1061	Yes	17.2	1161	Yes	19.1	1261	Yes	17.2	1361	Yes	21.1	1461	Yes	18.2
1062	?	17.5	1162	Yes	16.3	1262	?	18.2	1362	Yes	18.9	1462	?	13.1
1063	Yes	20.2	1163	Yes	10.6	1263	Yes	19.5	1363	Yes	20.5	1463	?	18.4
1064	Yes	22.6	1164	Yes	10.5	1264	?	18.6	1364	Yes	28.6	1464	Yes	15.7
1065	Yes	14.1	1165	Yes	12.3	1265	Yes	19.0	1365	Yes	30.6	1465	Yes	14.6
1066	Yes	14.1	1166	Yes	12.7	1266	?	19.5	1366	Yes	31.4	1466	Yes	16.4
1067	Yes	16.0	1167	Yes	13.3	1267	Yes	22.9	1367	Yes	31.7	1467	Yes	18.9
1068	?	17.6	1168	Yes	12.5	1268	Yes	22.2	1368	Yes	14.3	1468	Yes	20.7
1069	Yes	18.4	1169	?	12.4	1269	Yes	23.0	1369	Yes	16.1	1469	Yes	19.2
1070	Yes	25.8	1170	Yes	13.9	1270	Yes	17.1	1370	Yes	14.6	1470	Yes	20.7
1071	Yes	13.0	1171	Yes	12.1	1271	Yes	20.0	1371	Yes	15.4	1471	Yes	19.5
1072	Yes	18.1	1172	Yes	13.8	1272	Yes	20.8	1372	Yes	14.7	1472	Yes	19.7
1073	Yes	16.1	1173	Yes	13.5	1273	?	27.4	1373	Yes	16.5	1473	?	22.7
1074	Yes	18.7	1174	?	12.1	1274	Yes	19.2	1374	Yes	19.8	1474	Yes	18.5
1075	Yes	27.8	1175	Yes	15.9	1275	Yes	28.8	1375	Yes	21.1	1475	Yes	16.8
1076	Yes	12.7	1176	Yes	13.9	1276	Yes	22.7	1376	Yes	19.2	1476	Yes	16.6
1077	Yes	18.1	1177	Yes	19.4	1277	Yes	24.9	1377	Yes	19.0	1477	Yes	28.4
1078	Yes	18.4	1178	Yes	16.0	1278	?	25.2	1378	Yes	33.7	1478	Yes	29.5
1079	Yes	21.3	1179	Yes	16.4	1279	Yes	26.6	1379	Yes	34.6	1479	Yes	29.0
1080	Yes	26.9	1180	Yes	27.6	1280	Yes	19.7	1380	Yes	16.9	1480	Yes	27.8
1081	Yes	17.8	1181	Yes	23.9	1281	?	25.4	1381	Yes	19.0	1481	Yes	24.2
1082	Yes	12.8	1182	Yes	16.0	1282	Yes	26.2	1382	Yes	20.7	1482	Yes	26.4
1083	Yes	16.0	1183	Yes	19.4	1283	Yes	22.0	1383	Yes	21.9	1483	Yes	15.3
1084	Yes	19.5	1184	Yes	23.0	1284	Yes	21.3	1384	Yes	11.8	1484	Yes	16.6
1085	Yes	10.0	1185	Yes	21.1	1285	?	24.7	1385	?	17.3	1485	Yes	15.0
1086	Yes	11.0	1186	Yes	11.5	1286	Yes	22.6	1386	Yes	20.6	1486	Yes	12.6
1087	Yes	13.5	1187	Yes	11.9	1287	Yes	27.0	1387	?	12.3	1487	Yes	15.8
1088	Yes	13.5	1188	Yes	13.8	1288	Yes	23.9	1388	?	14.2	1488	Yes	20.1
1089	Yes	10.0	1189	?	18.7	1289	Yes	24.9	1389	Yes	14.1	1489	Yes	19.6
1090	Yes	11.8	1190	Yes	15.9	1290	Yes	32.1	1390	?	18.7	1490	Yes	17.3
1091	Yes	12.1	1191	Yes	14.6	1291	Yes	20.8	1391	?	19.2	1491	Yes	22.1
1092	Yes	12.0	1192	?	16.2	1292	?	15.1	1392	Yes	13.0	1492	Yes	16.4
1093	Yes	16.9	1193	Yes	12.3	1293	Yes	16.3	1393	Yes	14.3	1493	Yes	18.4
1094	Yes	19.5	1194	?	15.2	1294	Yes	14.2	1394	Yes	14.3	1494	?	18.5
1095	Yes	18.1	1195	Yes	14.6	1295	Yes	16.5	1395	?	12.5	1495	?	16.0
1096	Yes	19.8	1196	Yes	15.6	1296	Yes	20.0	1396	?	14.3	1496	?	19.1
1097	Yes	19.1	1197	Yes	18.0	1297	Yes	18.9	1397	Yes	13.8	1497	Yes	24.2
1098	Yes	12.9	1198	Yes	17.9	1298	Yes	15.5	1398	Yes	17.3	1498	Yes	24.2
1099	Yes	15.8	1199	Yes	21.0	1299	Yes	15.0	1399	Yes	22.0	1499	Yes	25.0
1100	Yes	21.3	1200	Yes	17.5	1300	Yes	15.6	1400	Yes	13.8	1500	Yes	19.5

#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time
1501	Yes	20.4	1601	Yes	19.9	1701	?	15.9	1801	Yes	30.3	1901	Yes	35.0
1502	Yes	26.1	1602	Yes	18.2	1702	Yes	18.7	1802	Yes	25.7	1902	Yes	14.8
1503	Yes	30.3	1603	Yes	20.2	1703	Yes	17.2	1803	Yes	28.5	1903	?	16.9
1504	Yes	29.8	1604	Yes	17.3	1704	?	18.4	1804	Yes	27.8	1904	Yes	19.6
1505	Yes	28.1	1605	Yes	23.6	1705	Yes	16.2	1805	Yes	29.1	1905	Yes	19.5
1506	Yes	29.2	1606	Yes	21.9	1706	Yes	17.7	1806	Yes	29.0	1906	Yes	19.9
1507	Yes	27.3	1607	Yes	33.8	1707	Yes	19.6	1807	Yes	28.0	1907	Yes	20.0
1508	Yes	23.2	1608	Yes	13.8	1708	Yes	18.3	1808	Yes	34.1	1908	Yes	21.7
1509	Yes	24.8	1609	Yes	17.5	1709	Yes	18.8	1809	Yes	40.5	1909	Yes	19.4
1510	Yes	23.7	1610	Yes	19.2	1710	Yes	20.4	1810	Yes	13.1	1910	Yes	20.7
1511	Yes	33.3	1611	Yes	19.8	1711	Yes	27.0	1811	Yes	21.7	1911	Yes	24.5
1512	Yes	38.1	1612	Yes	14.0	1712	Yes	16.2	1812	Yes	16.6	1912	Yes	24.3
1513	Yes	16.0	1613	Yes	18.3	1713	Yes	16.3	1813	Yes	15.9	1913	Yes	27.0
1514	Yes	14.9	1614	Yes	15.5	1714	Yes	22.8	1814	Yes	13.3	1914	Yes	24.8
1515	Yes	14.4	1615	?	16.8	1715	Yes	18.6	1815	Yes	19.9	1915	Yes	24.2
1516	Yes	13.8	1616	Yes	12.1	1716	Yes	12.8	1816	Yes	22.0	1916	Yes	25.6
1517	Yes	12.9	1617	Yes	14.0	1717	Yes	11.4	1817	Yes	13.7	1917	Yes	24.4
1518	Yes	17.5	1618	Yes	15.3	1718	Yes	12.6	1818	Yes	19.7	1918	Yes	21.8
1519	Yes	19.8	1619	Yes	12.2	1719	Yes	14.5	1819	Yes	18.1	1919	Yes	24.6
1520	Yes	15.2	1620	Yes	12.8	1720	Yes	14.8	1820	Yes	15.9	1920	Yes	23.3
1521	Yes	16.4	1621	Yes	13.7	1721	Yes	14.6	1821	Yes	16.6	1921	Yes	26.1
1522	Yes	17.5	1622	Yes	16.0	1722	Yes	18.3	1822	Yes	21.4	1922	Yes	28.7
1523	Yes	14.3	1623	?	17.8	1723	Yes	15.8	1823	Yes	27.7	1923	Yes	29.6
1524	Yes	12.6	1624	Yes	13.8	1724	Yes	19.0	1824	Yes	17.0	1924	Yes	33.0
1525	Yes	15.3	1625	?	17.8	1725	Yes	17.3	1825	Yes	16.1	1925	Yes	33.3
1526	?	16.3	1626	Yes	16.8	1726	Yes	20.9	1826	Yes	12.9	1926	Yes	40.7
1527	Yes	22.9	1627	Yes	15.7	1727	Yes	14.6	1827	Yes	14.7	1927	Yes	25.0
1528	Yes	19.9	1628	?	18.7	1728	Yes	15.4	1828	Yes	17.6	1928	Yes	18.0
1529	Yes	13.1	1629	Yes	20.1	1729	Yes	21.7	1829	Yes	20.5	1929	Yes	22.4
1530	?	15.9	1630	Yes	19.5	1730	Yes	20.4	1830	Yes	30.1	1930	Yes	20.2
1531	?	20.4	1631	Yes	18.2	1731	Yes	19.5	1831	Yes	13.5	1931	Yes	20.0
1532	Yes	23.1	1632	Yes	20.9	1732	?	22.4	1832	Yes	12.1	1932	Yes	22.8
1533	Yes	25.1	1633	Yes	19.6	1733	Yes	16.9	1833	Yes	14.6	1933	Yes	26.8
1534	Yes	29.0	1634	Yes	16.2	1734	Yes	16.4	1834	Yes	13.9	1934	Yes	22.1
1535	Yes	14.6	1635	Yes	19.3	1735	?	17.3	1835	Yes	15.3	1935	Yes	19.9
1536	Yes	21.1	1636	Yes	17.2	1736	Yes	20.8	1836	Yes	17.7	1936	Yes	20.7
1537	Yes	11.7	1637	Yes	14.7	1737	?	16.9	1837	Yes	19.5	1937	?	15.1
1538	Yes	15.9	1638	Yes	19.5	1738	Yes	16.0	1838	Yes	14.6	1938	Yes	20.4
1539	Yes	19.0	1639	Yes	14.7	1739	Yes	20.3	1839	Yes	11.8	1939	Yes	17.5
1540	Yes	13.3	1640	Yes	22.5	1740	Yes	13.0	1840	Yes	14.8	1940	Yes	20.6
1541	Yes	14.8	1641	Yes	21.2	1741	Yes	16.4	1841	Yes	16.9	1941	Yes	20.4
1542	Yes	12.7	1642	Yes	16.2	1742	Yes	13.6	1842	Yes	16.3	1942	Yes	30.5
1543	Yes	15.3	1643	?	13.1	1743	?	16.8	1843	Yes	15.7	1943	Yes	26.3
1544	Yes	20.0	1644	Yes	17.6	1744	Yes	16.9	1844	?	17.6	1944	Yes	24.6
1545	Yes	14.3	1645	Yes	17.1	1745	Yes	18.1	1845	Yes	16.0	1945	Yes	28.6
1546	Yes	15.5	1646	Yes	18.5	1746	?	19.0	1846	Yes	17.1	1946	Yes	33.7
1547	Yes	19.5	1647	Yes	19.7	1747	Yes	18.6	1847	Yes	14.4	1947	Yes	29.6
1548	Yes	19.6	1648	Yes	17.7	1748	Yes	19.9	1848	Yes	18.6	1948	Yes	33.6
1549	Yes	19.4	1649	Yes	19.1	1749	Yes	19.4	1849	Yes	17.0	1949	Yes	27.0
1550	Yes	18.5	1650	Yes	21.0	1750	Yes	28.5	1850	Yes	16.0	1950	Yes	25.7
1551	Yes	16.5	1651	Yes	21.3	1751	Yes	24.3	1851	Yes	14.1	1951	Yes	27.9
1552	Yes	16.9	1652	Yes	20.7	1752	Yes	18.9	1852	Yes	12.8	1952	Yes	44.0
1553	Yes	17.9	1653	?	22.5	1753	Yes	22.7	1853	?	18.0	1953	Yes	12.6
1554	Yes	26.7	1654	Yes	26.8	1754	Yes	19.9	1854	Yes	19.1	1954	Yes	33.6
1555	Yes	24.0	1655	Yes	17.2	1755	Yes	15.6	1855	?	15.6	1955	Yes	17.4
1556	Yes	19.6	1656	Yes	18.2	1756	Yes	27.4	1856	Yes	18.9	1956	Yes	16.0
1557	Yes	20.2	1657	Yes	24.4	1757	Yes	24.4	1857	?	18.4	1957	Yes	19.5
1558	Yes	12.4	1658	Yes	25.1	1758	Yes	21.3	1858	Yes	22.5	1958	Yes	20.5
1559	Yes	17.1	1659	Yes	23.9	1759	Yes	22.3	1859	Yes	20.4	1959	Yes	20.0
1560	Yes	24.3	1660	Yes	18.7	1760	Yes	24.5	1860	Yes	25.5	1960	Yes	27.2
1561	Yes	15.9	1661	Yes	27.0	1761	Yes	25.4	1861	Yes	27.1	1961	Yes	34.9
1562	?	16.4	1662	Yes	29.6	1762	Yes	27.7	1862	Yes	22.9	1962	Yes	25.3
1563	Yes	22.3	1663	Yes	29.0	1763	Yes	25.2	1863	Yes	18.7	1963	Yes	17.0
1564	Yes	24.4	1664	Yes	26.4	1764	Yes	24.5	1864	Yes	20.3	1964	Yes	17.3
1565	Yes	28.8	1665	Yes	30.8	1765	Yes	33.0	1865	Yes	19.2	1965	Yes	15.9
1566	Yes	22.1	1666	Yes	28.6	1766	Yes	27.2	1866	Yes	26.4	1966	Yes	12.7
1567	Yes	13.9	1667	Yes	23.7	1767	Yes	24.9	1867	Yes	28.7	1967	Yes	17.5
1568	Yes	16.0	1668	Yes	47.2	1768	Yes	20.1	1868	Yes	27.7	1968	Yes	20.9
1569	Yes	14.5	1669	Yes	13.3	1769	Yes	19.5	1869	Yes	44.4	1969	Yes	14.6
1570	Yes	12.9	1670	Yes	18.5	1770	Yes	21.0	1870	Yes	24.2	1970	Yes	14.5
1571	Yes	21.0	1671	Yes	18.1	1771	Yes	25.5	1871	Yes	16.0	1971	Yes	17.7
1572	Yes	26.3	1672	Yes	30.7	1772	Yes	18.7	1872	Yes	19.8	1972	Yes	20.2
1573	Yes	12.0	1673	Yes	13.8	1773	Yes	15.5	1873	Yes	28.2	1973	Yes	22.4
1574	Yes	15.9	1674	Yes	12.4	1774	Yes	13.6	1874	Yes	14.0	1974	Yes	24.3
1575	Yes	16.2	1675	Yes	19.4	1775	Yes	13.9	1875	Yes	14.1	1975	Yes	16.0
1576	Yes	19.7	1676	Yes	20.3	1776	Yes	16.0	1876	Yes	14.8	1976	Yes	13.7
1577	Yes	19.6	1677	Yes	27.1	1777	Yes	19.7	1877	Yes	18.2	1977	Yes	16.4
1578	Yes	13.6	1678	Yes	16.6	1778	Yes	18.8	1878	Yes	17.1	1978	Yes	13.8
1579	Yes	14.4	1679	Yes	25.1	1779	Yes	20.4	1879	Yes	19.6	1979	Yes	14.0
1580	Yes	18.7	1680	Yes	25.1	1780	Yes	22.8	1880	Yes	30.4	1980	Yes	18.3
1581	Yes	11.9	1681	Yes	19.7	1781	Yes	16.9	1881	Yes	12.5	1981	Yes	16.0
1582	Yes	13.6	1682	Yes	28.7	1782	Yes	20.3	1882	?	17.6	1982	?	18.4
1583	Yes	16.1	1683	Yes	14.2	1783	Yes	20.3	1883	Yes	16.1	1983	Yes	18.8
1584	Yes	13.3	1684	Yes	15.8	1784	Yes	20.1	1884	Yes	15.0	1984	Yes	18.7
1585	Yes	10.8	1685	Yes	20.7	1785	Yes	19.4	1885	Yes	17.8	1985	Yes	28.5
1586	Yes	18.5	1686	Yes	15.0	1786	?	22.3	1886	Yes	16.1	1986	Yes	13.9
1587	Yes	13.3	1687	Yes	17.5	1787	Yes	22.1	1887	Yes	17.1	1987	Yes	15.5
1588	Yes	14.2	1688	Yes	12.2	1788	Yes	24.2	1888	Yes	18.5	1988	Yes	12.7
1589	Yes	12.7	1689	Yes	13.8	1789	Yes	25.3	1889	Yes	19.6	1989	Yes	16.5
1590	Yes	11.3	1690	Yes	14.8	1790	Yes	23.5	1890	Yes	18.1	1990	Yes	15.0
1591	Yes	14.9	1691	Yes	12.9	1791	Yes	26.2	1891	Yes	19.1	1991	Yes	15.8
1592	Yes	15.1	1692	Yes	11.4	1792	Yes	30.1	1892	Yes	18.5	1992	Yes	21.8
1593	Yes	11.9	1693	Yes	12.8	1793	Yes	26.6	1893	Yes	21.9	1993	Yes	26.0
1594	Yes	13.4	1694	Yes	12.9	1794	Yes	26.9	1894	Yes	20.7	1994	Yes	25.5
1595	Yes	16.2	1695	Yes	13.1	1795	Yes	24.0	1895	Yes	22.2	1995	Yes	22.3
1596	Yes	14.6	1696	Yes	16.1	1796	Yes	25.5	1896	Yes	25.7	1996	Yes	19.9
1597	Yes	14.4	1697	Yes	20.2	1797	Yes	18.6	1897	Yes	25.6	1997	Yes	17.4
1598	?	13.7	1698	Yes	12.6	1798	Yes	18.3	1898	Yes	24.6	1998	Yes	20.2
1599	Yes	17.7	1699	?	15.7	1799	Yes	24.6	1899	Yes	24.2	1999	Yes	23.7
1600	Yes	17.8	1700	Yes	14.3	1800	Yes	27.3	1900	Yes	29.0	2000	Yes	23.8

#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time	#	Inconsistent	Time
2001	Yes	22.9	2101	Yes	16.8	2201	Yes	14.2	2301	Yes	25.7	2401	Yes	12.5
2002	Yes	27.3	2102	Yes	15.7	2202	Yes	20.8	2302	Yes	31.5	2402	Yes	19.4
2003	Yes	22.7	2103	Yes	24.8	2203	Yes	13.9	2303	Yes	18.8	2403	Yes	18.1
2004	Yes	19.2	2104	Yes	17.6	2204	Yes	14.9	2304	Yes	24.1	2404	Yes	16.0
2005	Yes	14.2	2105	Yes	19.1	2205	Yes	20.5	2305	Yes	10.7	2405	Yes	20.5
2006	Yes	14.3	2106	Yes	20.0	2206	Yes	24.0	2306	Yes	14.2	2406	Yes	19.8
2007	Yes	19.3	2107	Yes	25.3	2207	Yes	18.3	2307	Yes	12.6	2407	Yes	17.5
2008	Yes	14.0	2108	Yes	24.4	2208	Yes	16.6	2308	Yes	19.2	2408	Yes	19.8
2009	Yes	17.3	2109	Yes	19.4	2209	Yes	18.4	2309	Yes	20.4	2409	Yes	20.4
2010	Yes	20.1	2110	Yes	18.9	2210	Yes	19.3	2310	Yes	13.8	2410	Yes	18.4
2011	Yes	15.6	2111	Yes	25.6	2211	Yes	24.3	2311	Yes	15.7	2411	?	21.6
2012	Yes	22.9	2112	Yes	34.3	2212	Yes	19.7	2312	Yes	14.9	2412	Yes	27.3
2013	Yes	14.0	2113	Yes	14.3	2213	?	22.5	2313	Yes	17.9	2413	Yes	22.3
2014	Yes	16.2	2114	Yes	15.4	2214	Yes	18.2	2314	?	15.6	2414	?	24.0
2015	Yes	19.7	2115	?	11.9	2215	Yes	20.3	2315	Yes	19.5	2415	Yes	20.6
2016	?	13.2	2116	?	12.3	2216	Yes	15.3	2316	Yes	16.9	2416	Yes	21.6
2017	Yes	17.9	2117	?	11.2	2217	Yes	20.7	2317	Yes	17.4	2417	Yes	25.9
2018	Yes	23.0	2118	Yes	11.2	2218	Yes	20.8	2318	Yes	21.5	2418	Yes	34.5
2019	Yes	24.5	2119	Yes	16.0	2219	Yes	22.5	2319	Yes	18.5	2419	Yes	29.4
2020	Yes	32.3	2120	Yes	13.8	2220	?	21.9	2320	Yes	17.5	2420	Yes	19.4
2021	Yes	17.7	2121	Yes	17.7	2221	Yes	17.3	2321	Yes	19.6	2421	Yes	17.9
2022	Yes	19.2	2122	Yes	18.5	2222	Yes	17.8	2322	Yes	20.0	2422	Yes	23.8
2023	Yes	23.3	2123	Yes	19.8	2223	Yes	19.8	2323	Yes	25.1	2423	Yes	17.5
2024	Yes	24.4	2124	?	19.0	2224	Yes	21.5	2324	Yes	27.2	2424	Yes	19.7
2025	Yes	19.3	2125	?	19.1	2225	Yes	17.3	2325	Yes	20.5	2425	Yes	23.2
2026	Yes	19.0	2126	Yes	23.0	2226	Yes	22.5	2326	Yes	24.2	2426	Yes	23.1
2027	Yes	24.7	2127	?	19.7	2227	Yes	17.2	2327	Yes	20.2	2427	Yes	24.7
2028	Yes	29.1	2128	Yes	26.4	2228	Yes	19.3	2328	Yes	28.3	2428	?	24.6
2029	Yes	25.7	2129	Yes	27.3	2229	Yes	24.8	2329	Yes	26.0	2429	Yes	27.8
2030	Yes	22.3	2130	Yes	17.5	2230	Yes	22.4	2330	Yes	13.6	2430	Yes	29.4
2031	Yes	18.2	2131	Yes	18.2	2231	Yes	19.0	2331	Yes	17.9	2431	Yes	27.9
2032	Yes	27.9	2132	?	11.5	2232	Yes	27.7	2332	Yes	17.2	2432	Yes	27.3
2033	Yes	19.0	2133	Yes	12.7	2233	Yes	25.1	2333	Yes	17.1	2433	Yes	34.1
2034	Yes	20.7	2134	Yes	14.9	2234	Yes	24.6	2334	Yes	18.4	2434	Yes	27.0
2035	Yes	21.5	2135	Yes	14.4	2235	Yes	27.0	2335	Yes	19.1	2435	Yes	32.2
2036	Yes	28.1	2136	Yes	14.7	2236	Yes	22.1	2336	Yes	22.0	2436	Yes	26.3
2037	Yes	25.3	2137	Yes	15.2	2237	Yes	23.3	2337	?	18.3	2437	Yes	37.6
2038	Yes	33.8	2138	Yes	16.0	2238	Yes	27.6	2338	?	17.4	2438	Yes	28.8
2039	Yes	26.4	2139	?	13.4	2239	Yes	37.3	2339	Yes	20.7	2439	Yes	26.5
2040	Yes	14.4	2140	?	14.2	2240	Yes	26.1	2340	Yes	19.5	2440	Yes	27.4
2041	Yes	17.4	2141	?	14.8	2241	Yes	34.8	2341	Yes	22.4	2441	Yes	24.1
2042	Yes	19.6	2142	?	15.2	2242	Yes	23.2	2342	Yes	24.7	2442	Yes	35.7
2043	Yes	26.3	2143	?	13.9	2243	Yes	29.7	2343	Yes	20.5	2443	Yes	36.1
2044	Yes	44.4	2144	Yes	16.1	2244	Yes	25.2	2344	Yes	23.0	2444	Yes	34.2
2045	Yes	23.7	2145	Yes	15.4	2245	Yes	27.7	2345	Yes	23.6	2445	Yes	31.5
2046	Yes	13.7	2146	Yes	12.8	2246	Yes	23.8	2346	Yes	19.4	2446	Yes	35.2
2047	Yes	11.5	2147	Yes	13.1	2247	Yes	22.7	2347	Yes	26.0	2447	Yes	19.8
2048	?	11.5	2148	Yes	15.6	2248	Yes	18.5	2348	Yes	20.6	2448	Yes	24.3
2049	?	14.1	2149	Yes	13.6	2249	Yes	35.2	2349	Yes	15.1	2449	Yes	27.2
2050	Yes	17.1	2150	Yes	19.1	2250	Yes	14.0	2350	?	20.7	2450	Yes	22.9
2051	Yes	13.3	2151	Yes	18.4	2251	Yes	13.1	2351	?	20.6	2451	Yes	30.8
2052	Yes	13.8	2152	Yes	13.8	2252	Yes	16.6	2352	Yes	14.6	2452	Yes	25.1
2053	Yes	20.1	2153	Yes	16.3	2253	Yes	25.3	2353	Yes	19.2	2453	Yes	25.9
2054	Yes	19.2	2154	Yes	18.3	2254	Yes	22.9	2354	Yes	23.0	2454	Yes	17.1
2055	Yes	17.7	2155	Yes	20.0	2255	Yes	16.0	2355	Yes	21.9	2455	Yes	16.2
2056	?	12.6	2156	?	17.5	2256	Yes	26.6	2356	Yes	22.8	2456	Yes	19.6
2057	Yes	12.0	2157	?	18.8	2257	Yes	18.3	2357	Yes	15.9	2457	Yes	19.0
2058	Yes	11.8	2158	Yes	16.3	2258	Yes	13.3	2358	Yes	17.8	2458	Yes	19.9
2059	Yes	19.5	2159	Yes	20.5	2259	Yes	24.6	2359	Yes	27.2	2459	Yes	22.6
2060	Yes	14.2	2160	?	17.8	2260	Yes	14.2	2360	Yes	20.4	2460	Yes	15.8
2061	Yes	11.3	2161	Yes	19.9	2261	Yes	13.1	2361	Yes	16.5	2461	Yes	17.1
2062	Yes	12.0	2162	Yes	17.0	2262	Yes	12.5	2362	Yes	18.3	2462	Yes	19.6
2063	?	15.3	2163	Yes	23.8	2263	Yes	17.2	2363	Yes	19.0	2463	Yes	18.9
2064	Yes	18.2	2164	?	16.9	2264	Yes	21.7	2364	Yes	18.2	2464	Yes	26.3
2065	?	12.3	2165	?	18.0	2265	Yes	24.4	2365	Yes	27.3	2465	Yes	17.4
2066	?	15.8	2166	Yes	27.3	2266	Yes	18.3	2366	Yes	27.1	2466	Yes	19.1
2067	?	15.0	2167	Yes	20.5	2267	Yes	30.0	2367	Yes	25.1	2467	Yes	20.2
2068	Yes	13.1	2168	Yes	24.5	2268	Yes	17.9	2368	Yes	28.9	2468	Yes	23.4
2069	Yes	19.6	2169	Yes	19.0	2269	Yes	15.9	2369	Yes	20.2	2469	Yes	34.3
2070	Yes	14.1	2170	?	18.9	2270	Yes	24.5	2370	Yes	24.2	2470	Yes	29.7
2071	Yes	19.3	2171	Yes	17.4	2271	Yes	25.4	2371	Yes	18.9	2471	Yes	24.3
2072	Yes	14.7	2172	?	17.4	2272	Yes	26.1	2372	?	22.5	2472	Yes	19.2
2073	Yes	14.9	2173	Yes	18.2	2273	Yes	37.5	2373	Yes	21.2	2473	Yes	21.3
2074	Yes	15.3	2174	Yes	12.6	2274	Yes	27.9	2374	Yes	23.3	2474	Yes	18.6
2075	Yes	15.8	2175	Yes	14.8	2275	Yes	27.1	2375	Yes	25.4	2475	Yes	26.1
2076	Yes	20.9	2176	Yes	19.7	2276	Yes	31.9	2376	Yes	23.2	2476	Yes	23.0
2077	Yes	22.7	2177	Yes	16.7	2277	Yes	24.8	2377	Yes	30.5	2477	Yes	22.1
2078	Yes	17.2	2178	Yes	18.8	2278	Yes	22.6	2378	Yes	36.2	2478	Yes	19.3
2079	Yes	16.4	2179	Yes	21.2	2279	Yes	33.2	2379	Yes	26.0	2479	Yes	21.8
2080	Yes	20.0	2180	Yes	22.7	2280	Yes	18.1	2380	Yes	36.0	2480	Yes	20.6
2081	Yes	16.3	2181	Yes	16.1	2281	Yes	24.0	2381	Yes	23.9	2481	Yes	26.1
2082	Yes	18.9	2182	Yes	21.5	2282	Yes	16.7	2382	Yes	28.9	2482	Yes	23.1
2083	Yes	22.4	2183	Yes	20.0	2283	Yes	10.6	2383	Yes	30.0	2483	Yes	27.5
2084	?	18.2	2184	Yes	19.1	2284	?	10.9	2384	Yes	41.4	2484	Yes	33.4
2085	Yes	21.5	2185	Yes	25.5	2285	Yes	14.3	2385	Yes	18.6	2485	Yes	32.3
2086	Yes	20.4	2186	Yes	20.0	2286	Yes	17.7	2386	Yes	22.4	2486	Yes	16.9
2087	Yes	19.8	2187	?	20.2	2287	Yes	15.3	2387	Yes	13.6	2487	Yes	17.4
2088	Yes	17.0	2188	Yes	26.9	2288	Yes	14.2	2388	Yes	16.0	2488	Yes	19.3
2089	Yes	19.8	2189	Yes	23.2	2289	Yes	15.5	2389	Yes	17.4	2489	Yes	13.4
2090	Yes	20.2	2190	Yes	22.8	2290	Yes	18.2	2390	Yes	20.3	2490	Yes	16.5
2091	Yes	22.6	2191	Yes	26.2	2291	Yes	16.8	2391	Yes	18.2	2491	Yes	13.1
2092	Yes	25.9	2192	Yes	17.5	2292	Yes	19.2	2392	?	16.4	2492	?	16.8
2093	Yes	21.2	2193	?	14.3	2293	Yes	18.4	2393	Yes	18.0	2493	Yes	15.0
2094	Yes	22.2	2194	Yes	18.6	2294	Yes	24.5	2394	?	17.9	2494	Yes	11.2
2095	Yes	16.0	2195	Yes	17.4	2295	Yes	17.2	2395	Yes	15.9	2495	Yes	10.4
2096	Yes	17.3	2196	Yes	17.1	2296	Yes	17.0	2396	Yes	18.7	2496	Yes	13.6
2097	Yes	19.9	2197	Yes	14.6	2297	Yes	17.8	2397	Yes	19.9	2497	Yes	13.5
2098	Yes	14.0	2198	Yes	14.8	2298	Yes	29.3	2398	Yes	24.1	2498	Yes	15.8
2099	Yes	15.0	2199	Yes	18.4	2299	Yes	26.2	2399	Yes	29.4	2499	Yes	14.5
2100	Yes	15.8	2200	Yes	17.1	2300	Yes	16.9	2400	Yes	16.7	2500	Yes	13.8

#	Inconsistent	Time
2501	Yes	19.2
2502	Yes	22.9
2503	Yes	20.5
2504	Yes	17.3
2505	Yes	18.4
2506	Yes	32.5
2507	Yes	11.5
2508	Yes	11.2
2509	Yes	14.2
2510	Yes	13.6
2511	Yes	16.6
2512	Yes	10.7
2513	Yes	12.8
2514	Yes	14.0
2515	Yes	17.4
2516	Yes	16.6
2517	Yes	17.1
2518	Yes	17.1
2519	?	18.1
2520	Yes	21.0
2521	Yes	24.1
2522	Yes	27.1
2523	Yes	19.9
2524	Yes	24.6
2525	Yes	20.4
2526	Yes	33.6
2527	Yes	20.2
2528	Yes	19.2
2529	Yes	20.6
2530	?	23.3
2531	Yes	21.3
2532	Yes	27.9
2533	Yes	28.0
2534	Yes	28.0
2535	Yes	27.9
2536	Yes	26.5
2537	Yes	12.2
2538	Yes	11.7
2539	Yes	14.0
2540	Yes	14.2
2541	Yes	19.7
2542	Yes	14.3
2543	Yes	14.4
2544	Yes	10.6
2545	Yes	11.1
2546	Yes	13.2
2547	?	12.1
2548	Yes	13.0
2549	Yes	13.7
2550	Yes	12.6
2551	?	13.3
2552	Yes	16.5
2553	Yes	14.6
2554	Yes	18.6
2555	Yes	14.7
2556	Yes	19.6
2557	Yes	18.6
2558	Yes	19.0
2559	Yes	16.8
2560	Yes	19.0
2561	Yes	19.9
2562	Yes	15.4
2563	Yes	27.0
2564	?	17.6
2565	?	16.6
2566	?	19.2
2567	Yes	23.6
2568	Yes	23.6
2569	Yes	18.7
2570	Yes	17.9
2571	Yes	25.5
2572	Yes	22.2
2573	Yes	22.8
2574	Yes	19.6
2575	Yes	34.1
2576	Yes	17.1
2577	?	17.5
2578	Yes	19.4
2579	Yes	16.1
2580	Yes	15.6
2581	Yes	19.9
2582	Yes	22.1
2583	Yes	25.1
2584	Yes	25.3
2585	Yes	25.0
2586	Yes	33.7
2587	Yes	31.3
2588	Yes	27.7
2589	Yes	28.0
2590	Yes	18.0
2591	Yes	20.4
2592	Yes	20.5
2593	Yes	29.2
2594	Yes	30.9
2595	Yes	13.7
2596	Yes	15.3
2597	Yes	14.3
2598	Yes	17.0
2599	Yes	12.9
2600	Yes	15.3

#	Inconsistent	Time
2601	Yes	18.3
2602	Yes	14.9
2603	?	14.5
2604	Yes	13.3
2605	Yes	13.5
2606	Yes	24.4
2607	?	20.6
2608	Yes	14.3
2609	Yes	14.4
2610	Yes	15.5
2611	?	16.9
2612	?	20.9
2613	?	17.1
2614	?	21.1
2615	Yes	17.2
2616	Yes	19.6
2617	?	18.5
2618	Yes	18.7
2619	Yes	23.6
2620	Yes	19.3
2621	Yes	14.0
2622	Yes	14.7
2623	?	17.9
2624	Yes	19.9
2625	Yes	24.0
2626	Yes	15.7
2627	Yes	13.4
2628	Yes	16.8
2629	Yes	13.2
2630	?	17.0
2631	Yes	15.7
2632	Yes	14.1
2633	Yes	19.6
2634	Yes	21.0
2635	Yes	14.5
2636	Yes	19.6
2637	Yes	14.6
2638	?	16.8
2639	Yes	25.8
2640	?	19.9
2641	Yes	13.6
2642	Yes	13.0
2643	?	16.9
2644	?	17.8
2645	Yes	17.2
2646	Yes	18.5
2647	Yes	22.0
2648	Yes	14.7
2649	?	17.3
2650	Yes	12.5
2651	Yes	20.2
2652	Yes	24.8
2653	?	20.3
2654	Yes	14.1
2655	Yes	24.7
2656	Yes	14.3
2657	Yes	19.3
2658	Yes	25.9
2659	?	20.4
2660	Yes	14.0
2661	?	20.4
2662	Yes	12.4
2663	Yes	13.0
2664	Yes	20.2
2665	Yes	13.3
2666	Yes	17.8
2667	Yes	15.6
2668	Yes	13.5
2669	Yes	13.7
2670	?	13.4
2671	Yes	17.4
2672	Yes	16.9
2673	Yes	12.7
2674	?	17.7
2675	?	16.8
2676	Yes	19.3
2677	?	17.9
2678	Yes	22.9
2679	Yes	25.5
2680	Yes	13.8
2681	?	17.5
2682	?	16.7
2683	Yes	20.3
2684	Yes	16.6
2685	Yes	18.3
2686	Yes	20.1
2687	Yes	13.3
2688	?	17.5
2689	Yes	20.2
2690	Yes	20.2
2691	Yes	24.1
2692	Yes	25.3
2693	Yes	24.0
2694	Yes	23.4
2695	Yes	27.1
2696	Yes	15.6
2697	Yes	13.3
2698	Yes	16.5
2699	Yes	20.4
2700	Yes	25.8

#	Inconsistent	Time
2701	Yes	14.3
2702	?	16.9
2703	Yes	18.0
2704	Yes	18.8
2705	Yes	19.6
2706	Yes	16.9
2707	Yes	13.7
2708	Yes	16.4
2709	Yes	20.7
2710	Yes	24.0
2711	Yes	25.0
2712	Yes	28.6
2713	Yes	22.8
2714	Yes	13.5
2715	Yes	21.9
2716	Yes	16.4
2717	Yes	28.6
2718	?	18.9
2719	Yes	18.5
2720	Yes	20.1
2721	Yes	16.2
2722	Yes	26.3
2723	Yes	23.4
2724	Yes	12.6
2725	?	16.0
2726	Yes	16.9
2727	Yes	13.4
2728	?	16.5
2729	?	19.0
2730	Yes	13.2
2731	?	16.9
2732	Yes	19.4
2733	Yes	13.8
2734	?	16.9
2735	Yes	14.8
2736	Yes	15.8
2737	Yes	14.9
2738	Yes	15.6
2739	Yes	14.4
2740	Yes	14.1
2741	Yes	15.3
2742	Yes	20.2
2743	Yes	15.3
2744	Yes	15.3
2745	Yes	15.0
2746	Yes	15.3
2747	Yes	19.1
2748	Yes	15.0
2749	Yes	18.1
2750	Yes	14.3
2751	Yes	17.9
2752	Yes	21.5
2753	Yes	15.5
2754	Yes	14.2
2755	Yes	14.6
2756	Yes	14.3
2757	Yes	17.6
2758	Yes	14.7
2759	Yes	20.8
2760	?	17.9
2761	Yes	14.9
2762	Yes	15.3
2763	Yes	21.7
2764	Yes	12.3
2765	Yes	12.1
2766	Yes	12.3
2767	Yes	11.8
2768	Yes	11.7
2769	Yes	11.8
2770	Yes	11.3
2771	Yes	13.1

List of Figures

Bibliography

[1] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.

[2] Bruno Barras. Programming and computing in hol. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 17–37. Springer, 2000.

[3] Gertrud Bauer. *Formalizing plane graph theory*. PhD thesis, Technische Universität München, 2006.

[4] Garrett Birkhoff. *Lattice Theory*. AMS, 1967.

[5] Dave Cohen and Phil Watson. An efficient representation of arithmetic for term rewriting. In Ronald V. Book, editor, *RTA*, volume 488 of *Lecture Notes in Computer Science*, pages 240–251. Springer, 1991.

[6] Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.

[7] László Fuchs. *Partially ordered algebraic systems*. Addison-Wesley, 1963.

[8] Georges Gonthier. A computer-checked proof of the Four Color Theorem. `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[9] Mike Gordon. From LCF to HOL: a short history. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

[10] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.

[11] Thomas C. Hales. Computer resources for the Kepler Conjecture. `http://annals.math.princeton.edu/keplerconjecture/`.

[12] Thomas C. Hales. Some algorithms arising in the proof of the kepler conjecture. sect. 3.1.1., arXiv:math.MG/0205209.

[13] Thomas C. Hales. Sphere packings III. arXiv:math/9811075v2, `http://arxiv.org/abs/math/9811075v2`.

[14] Thomas C. Hales. Introduction to the flyspeck project. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[15] Thomas C. Hales and Samuel P. Ferguson. A proof of the Kepler Conjecture. *Annals of Mathematics*, 162:1065–1185, 2005.

[16] Thomas C. Hales and Samuel P. Ferguson. The Kepler Conjecture. *Discrete & Computational Geometry*, 36, 2006.

[17] Joe Hurd and Thomas F. Melham, editors. *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*. Springer, 2005.

[18] Bart Jacobs and Thomas F. Melham. Translating dependent type theory into higher order logic. In *Typed Lambda Calculus and Applications*, pages 209–229, 1993.

[19] Alexander Krauss. Partial recursive functions in higher-order logic. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer, 2006.

[20] Serge Lang. *Algebra*. Addison-Wesley, 1974.

[21] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[22] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. The archive of tame graphs. `http://www4.informatik.tu-muenchen.de/~nipkow/pubs/Flyspeck`, 2006.

[23] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame graphs. In *IJCAR*, pages 21–35, 2006.

[24] Tobias Nipkow and Lawrence C. Paulson. Proof pearl: Defining functions over finite sets. In Hurd and Melham [17], pages 385–396.

[25] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[26] Steven Obua. Proving bounds for real linear programs in Isabelle/Hol. In Hurd and Melham [17], pages 227–244.

[27] Steven Obua. Checking conservativity of overloaded definitions in higher-order logic. In Frank Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2006.

[28] Steven Obua. Proof pearl: Looping around the orbit. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 223–231. Springer, 2007.

[29] Lawrence C. Paulson. *ML for the working programmer (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1996.

[30] Lawrence C. Paulson. Organizing numerical theories using axiomatic type classes. *J. Autom. Reasoning*, 33(1):29–49, 2004.

[31] Lawrence C. Paulson. Defining functions on equivalence classes. *ACM Trans. Comput. Log.*, 7(4):658–675, 2006.

[32] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, June 1998.

[33] Scott Owens and Konrad Slind. Adapting functional programs to higher order logic. *Higher Order and Symbolic Computation*. To appear.

[34] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy P. Felty, editors, *TPHOLs*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997.