Ann Math Artif Intell (2009) 56:245–272
DOI 10.1007/s10472-009-9168-z

Flyspeck II: the basic linear programs

Steven Obua · Tobias Nipkow

Published online: 7 November 2009
© Springer Science + Business Media B.V. 2009

Abstract We present another step, Flyspeck II, towards a complete, formal and mechanized proof of the Kepler Conjecture.

Keywords Flyspeck · Kepler conjecture · Interactive theorem proving · Isabelle · Higher-order logic · Computational logic · Finite matrices · Lattice-ordered rings · Linear programming · Hypermaps · Planar graphs

Mathematics Subject Classifications (2000) 03B35 · 68T15

1 The Flyspeck project

In 1998, the Kepler conjecture has been proven by Thomas Hales after remaining the most famous open conjecture in discrete geometry for about 400 years. Since then, both abbreviated [14] and more detailed [15] versions of the proof have been published. But even the more detailed published proof falls short of a complete account of the matter. Part of the 1998 proof are computer programs and logs of runs of these programs, and no referee of the published proofs has even so much as looked at them.

Therefore in 2003 Thomas Hales initiated the Flyspeck project. Its goal is to formalize the proof of the Kepler conjecture in a mechanized proof assistant in order to achieve the highest possible standard of mathematical rigour known to mankind.

Linear programming is an important part of the Kepler proof, there are several gigabytes of log files available as a result of running linear programs in the original

S. Obua (✉)
Universität des Saarlands, 66041, Saarbrücken, Germany
e-mail: steven@obua.com, obua@wjpserver.cs.uni-sb.de

T. Nipkow
TU München, Boltzmannstr. 3, 85748, Garching, Germany
e-mail: nipkow@in.tum.de

proof. In this article we present the second major part of the Flyspeck project, the formalization of the creation and bounding of these linear programs in the mechanical proof assistant Isabelle/HOL [25]. Most of what we describe in this article can be found in greater detail in [29]. There have been other major advances in the Flyspeck project; for an overview see [16].

2 The Kepler conjecture, tame graphs, and linear programs

In this section we explain how the notion of basic linear programs comes up in the proof of the Kepler Conjecture.

The Kepler conjecture states that the best density one can hope for when packing infinitely many balls of radius 1 in three-dimensional euclidean space is

$$\frac{\pi}{\sqrt{18}} \approx 0.74 \tag{1}$$

A packing that achieves this density is the *face-centered cubic packing*. This packing consists necessarily of infinitely many balls. Figure 1 shows a local section of this packing consisting of 13 spheres.

It is enough to consider only such local arrangements of spheres for the proof of the Kepler conjecture. That this is so is a consequence of the entire proof, not the other way around.

The systematic way to transition from a packing consisting of infinitely many balls to a local arrangement of only finitely many balls is to 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 $2\ t_0 = 2.51$ [15, p. 26]. Doing so leads from the face-centered cubic packing to the arrangement of spheres in Fig. 1.

Connect then all such centers with straight lines if their distance is less than $2\ t_0$. The left picture in Fig. 2 shows the result; we show only the origin ball, the remaining 12 spheres are represented by their centers only.

Now project all of the straight lines onto the ball at the origin, and you get the spherical plane graph depicted in the right picture of Fig. 2.

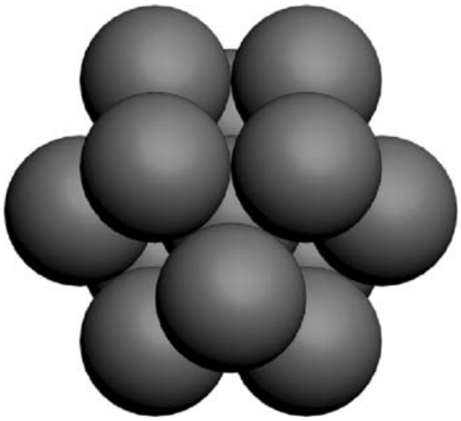
Fig. 1 Three layers of the face-centered cubic packing

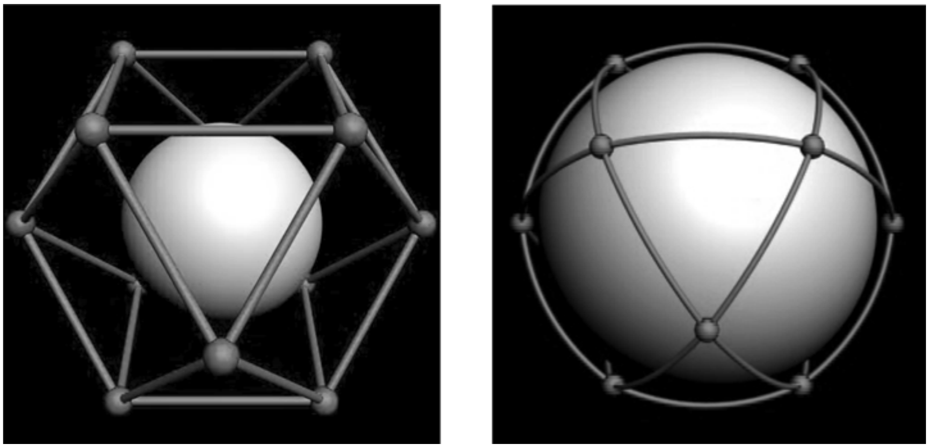

Fig. 2 The face-centered cubic packing as graphs in space and on the sphere

Choosing an interior point of the upper quadrilateral and applying stereographic projection transforms the spherical plane graph into the plane graph shown in Fig. 3.

The specific notion of *tameness* captures all plane graphs that can arise from packings which are dense enough to be candidates for packings of maximal density. Flyspeck I [23] has been concerned with formalizing the notion of tameness and producing an *archive* of 2771 plane graphs. This archive has been formally verified to contain all tame graphs. This implies that it has been formally verified that there are only finitely many tame graphs. This is one step of reducing the a-priori infinite-dimensional optimization problem of finding all packings of optimal density to a finite-dimensional one.

In the following we use the expression *tame graph* to describe any graph that is contained in the archive.

Hales's proof shows that optimal packings consist only of local configurations of spheres which correspond to tame graphs. To complete the proof of the Kepler

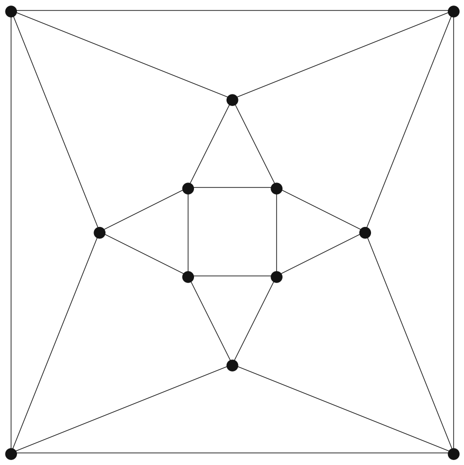

Fig. 3 The face-centered cubic packing as plane graph

conjecture, one has to further reduce the number of tame graphs that correspond to possibly optimal packings until only two graphs are left, the graph in Fig. 3 induced by the face-centered cubic packing, and the graph induced by the hexagonal close packing. The local arrangement of the hexagonal close packing can be obtained from the one in Fig. 1 by rotating the barely visible three balls of the lower layer by 60° such that they are completely hidden by the three balls of the upper layer.

To carry out the reduction we relate the tame graphs, which carry only topological information, back to the metric graphs in space (left picture in Fig. 2) as well as on the sphere (right picture in Fig. 2). For the sake of distinguishing the two graphs, let us call the metric graph in space the *euclidean graph* of the tame graph, and the metric graph on the sphere its *spherical graph*. Note that this nomenclatura is somewhat misleading because there are not necessarily unique euclidean and spherical graphs corresponding to a tame graph; two different local arrangements of spheres may induce the same tame graph.

Using the relation between tame, euclidean, and spherical graphs, certain real-valued variables are introduced on tame graphs. Many of these variables have straightforward geometric interpretations. For example, for any edge e of the tame graph the variable $ye\ e$ denotes the length of the corresponding edge in the euclidean graph. Therefore $2 \leq ye\ e < 2\ t_0$ must hold. Similarly, for any face F of the tame graph the variable $sol\ F$ measures the size of the area of the corresponding surface in the spherical graph. Because the whole surface of a unit ball is 4π, we have $4\pi = \sum sol\ F$, where the sum is taken over all faces of the tame graph.

The faces of a tame graph induce a partitioning of the volume around the origin. The most important variable is the *score* $\sigma\ F$ of a face F. Its geometric interpretation is not so straightforward, but intuitively it relates to the density of the volume that corresponds to F. Therefore, in order for the tame graph to qualify to be induced by an optimal packing, the score cannot be lower than a certain threshold; in particular, the inequality

$$8 * pt \leq \sum \sigma\ F \tag{2}$$

must hold where $pt \approx 0.0553736$. The sum is taken over all faces F of the tame graph. Those tame graphs which fulfill this inequality are called *contravening*. The goal is to show that there are only two contravening tame graphs.

There are more such real-valued variables. A complete list of all real-valued variables on tame graphs we have used in our work appears in Section 6.

The reader might wonder why we call these entities real-valued *variables* instead of real-valued *constants*. For example, for a given tame graph and a given edge e, the real value $ye\ e$ surely is a constant? The answer is that while $ye\ e$ is constant for a euclidean graph of the tame graph, it is not necessarily constant for the tame graph but varies along with the multiple euclidean graphs that belong to a single tame graph.

So far we have only mentioned linear relationships between the different real-valued variables. But the default case is that relationships are highly non-linear. For example, let $E = \sum ye\ e$ where the sum is taken over all edges of some fixed face F. Certainly a very small E also leads to a very small $sol\ F$, but this is a non-linear relationship which also depends non-linearly on the distance of the euclidean nodes from the origin and so on.

The strategy of Hales proof of the Kepler conjecture is to list enough such relationships between the real-valued variables of a tame graph until an inconsistency with the assumption that the graph is contravening arises. The natural way to prove this inconsistency is to solve a non-linear optimization problem in over 150 variables. Unfortunately, there do not exist techniques yet for solving a generic problem of this size in any practically relevant timeframe. Thomas Hales solved it anyway by using his geometric intuition to relax the non-linear relationships yielding linear relationships which can be subjected to linear optimization techniques. We selected a subset of these linear relationships and call the linear optimization problems they lead to *basic linear programs*, following [15, Sect. 23.3]. The subset has been selected such that

- it is a superset of the linear relationships listed in [15, Sect. 23.3],
- it is large enough to prove the majority of tame graphs to be non-contravening,
- it is straightforward how to convert the linear relationships into linear optimization problems without using techniques like branch-and-bound.

3 The approach

In order to prove that most tame graphs are not contravening we used the same approach that Thomas Hales followed in his 1998 proof of the Kepler conjecture, adjusted to the fact that we need to have a gapless, mechanically verified proof of our result.

We first capture in the notion (this notion is specific to our work) of a *graph system* the linear relationships that are necessary conditions for tame graphs to be contravening. Essentially, a graph system is just a predicate *GraphSystemP G* on tame graphs *G*. The complete definition of graph systems is given in Section 6.

We then apply the graph system predicate to a given tame graph *G*. Our goal is to prove

$$\mathit{GraphSystemP}\ G \quad \Longrightarrow \quad \text{False} \tag{3}$$

as a theorem in the Isabelle theorem prover.

Because a graph system captures linear relationships between real-valued variables, we can prove

$$\mathit{GraphSystemP}\ G \quad \Longrightarrow \quad A * x \leq b \tag{4}$$

for an $n \times m$ matrix A, an $m \times 1$ matrix x, and an $n \times 1$ matrix b. The matrices A and b contain terms consisting of real constants and basic arithmetical operations like addition or division, for example terms like 0 or $(4\pi + 7)/3$. The matrix x is a vector containing real-valued variables of the tame graph G, like *ye* 3 or *sol* 2, where 3 is some index of an edge of G, and 2 is some index of a face of G. We describe our representation of matrices in Isabelle/HOL in Section 7.

For the proof of the above theorem we make heavy use of rewriting. Usually, Isabelle's Simplifier is the tool of choice for rewriting in Isabelle. But for our application, the Simplifier was found to be much too slow. Therefore the HOL Computing Library (HCL) has been developed, which has been used for all major rewriting jobs. We present the HCL in Section 4.

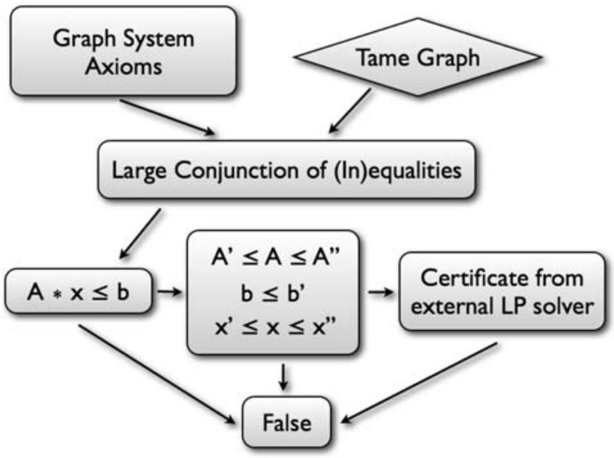

Fig. 4 Refuting a counter example to the Kepler conjecture (picture reproduced from [16])

The next step is to approximate the matrices A and b by other matrices A', A'' and b' such that these other matrices contain only concrete numerical values. The results of this approximation are theorems $A' \le A \le A''$ and $b \le b'$. From these theorems we can derive a-priori bounds x' and x'' for x such that $x' \le x \le x''$ holds [29, Sect. 3.8]. The approximation step again uses the HCL extensively.

From there we obtain via linear programming a certificate which allows us to prove in Isabelle

$$A * x \le b \quad \Longrightarrow \quad \text{False} \tag{5}$$

We describe this step in Section 7. Of course, checking the certificate again is based on the HCL.

Now the original goal (3) follows trivially. Figure 4 visualizes our approach.

4 The HOL computing library

The HOL Computing Library (HCL) is an extension of the Isabelle system for fast and trusted computing. It has been developed by the first author of this paper specifically for its application to the Flyspeck project. Nevertheless, the HCL is a general tool and *none* of its capabilities is there only to fulfill Flyspeck specific needs; *all* of its features are features you would want in a general purpose computing tool for HOL. A more detailed and in-depth description of the HCL than the one given in this section is available [29, Chapt. 2].

The architecture of the HCL and how it fits into the overall architecture of the Isabelle system is shown in Fig. 5.

The Isabelle theorem proving system follows a principle that has been pioneered by the Edinburgh LCF theorem prover [10]. Theorems are represented as an abstract datatype, and the logic is encoded as operations on this abstract datatype. The implementation of this abstract datatype is called the *kernel* of the Isabelle

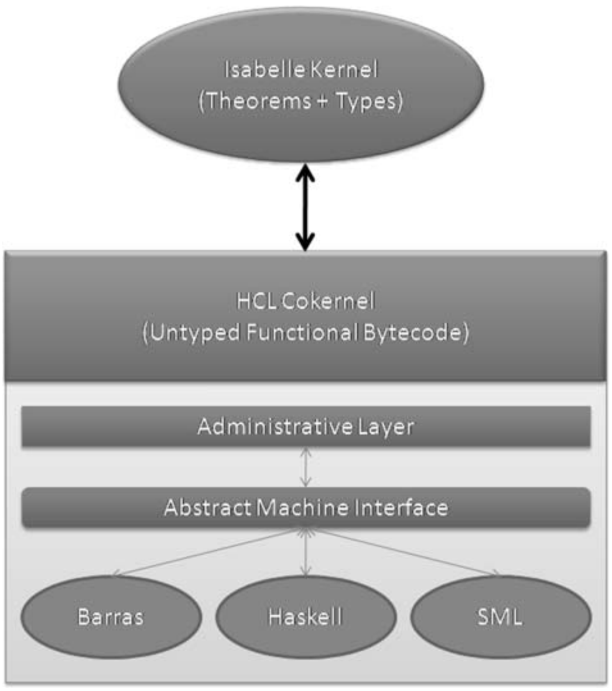

Fig. 5 Design of the HOL computing library (HCL)

system. Usually, Isabelle tools like decision procedures or the Simplifier live outside the kernel, manipulating theorems only through kernel operations. Even complex Isabelle tools built this way can therefore be trusted to the same degree as the relatively small kernel.

The Simplifier pays for its safety with its performance. It is much too slow for our purposes of computing with matrices of dimensions 2000×150 and more. Therefore the HCL bypasses the Isabelle kernel and manages its own internal representation of terms, theorems and programs. While there are proofs on paper [29, Chapt. 2] of key parts of the HCL, there is no formal mechanized proof of its correctness. Using the HCL means putting the same trust into the implementation of the HCL as into the implementation of the Isabelle kernel. A good analogy is to consider the HCL as a *cokernel* which sits besides the Isabelle kernel just as mathematical coprocessors used to accompany main processors.

The HCL consists of two components. Its *administrative layer* is responsible for the safe translation between Isabelle kernel data structures like terms and theorems and the corresponding HCL counter parts. For example, when Isabelle terms are passed to the HCL, the types are stripped from the term and (possibly overloaded) constants are encoded by integers. After computation, the resulting HCL term is converted back to an equivalent typed Isabelle term. The administrative layer also converts a list of Isabelle theorems into an HCL program. These Isabelle theorems must be conditional rewrite rules, as for example produced by the *function* package [20].

The other component is the *abstract machine interface* and its implementations. The abstract machine interface defines the rules for performing the actual computation, that is the rules for how to apply an HCL program to an HCL term. All abstract machine implementations must obey these rules. Currently there are three implementations available: The Barras mode, the Haskell mode, and the SML mode. For Flyspeck II, we use the SML mode, because it is the fastest. It translates the HCL program into a compiled Standard ML program. Similarly, the target of the Haskell mode is a compiled Haskell program. The Barras mode works by interpreting the HCL program using an evaluation strategy described in [3].

An advantage of this two-component architecture is that it decouples theorem proving issues from computing issues. The administrative layer is responsible for the safe embedding of the abstract machines into the theorem proving environment. The abstract machine layer knows nothing about theorems. Therefore a compiler technology expert who may know nothing about theorem proving technology could design a high-performance abstract machine implementation.

Table 1 (adapted from [29, Sect. 2.3]) shows a performance comparison between the Simplifier, the HCL, and a handwritten Standard ML program for computing the factorial $i!$ for increasing values of i. Note that the handwritten Standard ML program does not use native integers, but the same data structure for binary numbers as the Isabelle theory. The Simplifier and the HCL produce actual Isabelle theorems, the handwritten SML program just the resulting binary number. The runtimes have been scaled to the runtime of the SML program. Missing entries in the table do not indicate nontermination but just that these measurements have not been taken.

For small i, the administrative overhead of the HCL dominates the runtime. For large i, a clear trend emerges. The SML mode of the HCL is almost as fast as the handwritten SML program. The Haskell mode is about 5 times slower than the SML mode, and the Barras mode is about 90 times slower. And the slowdown factor of the Simplifier compared with the handwritten Standard ML program increases linearly with increasing i, becoming as large as 23000 for $i = 1280$!

The HCL is much faster than the Simplifier, but it is not as flexible. For example, the HCL has no simplification procedures, and it has no congruence rules, but only weaker conditional rules.

The reason why the HCL is nevertheless still very useful in a theorem proving environment is that the high degree of flexibility provided by the Simplifier is usually

Table 1 Performance comparison $\left(\frac{\text{runtime of method}}{\text{runtime for computing SML value}}\right)$

i	Simplifier	Barras	Haskell	SML	Standard ML
	Computing theorem				Computing SML value
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

not needed for computation. Instead, the HCL implements a few features which are useful when computing in a theorem prover:

Conditional rewrite rules Isabelle theorems which can be used as HCL rewrite rules for a function f have the general form

$$A_1 \equiv B_1 \Longrightarrow \ldots \Longrightarrow A_m \equiv B_m \Longrightarrow f\ p_1\ p_2 \ldots p_n \equiv T \tag{6}$$

The symbols $\equiv$ and $\Longrightarrow$ denote Isabelle's meta notions of equality and implication, respectively. Furthermore, $(p_1, p_2, \ldots p_n)$ must be a *linear pattern*. The body T and the conditions $A_i \equiv B_i$ may depend on variables bound in the p_i. The HCL only applies the rule if all its conditions hold. Condition i holds if after evaluating A_i and B_i the resulting terms are structurally equal.

Symbolic computation The HCL cannot only be used for raw computation like computing the factorial of some number but it can also do symbolic computation. There are two reasons for this. First, HCL programs are created from arbitrary conditional rewrite rules. Second, and most importantly, although the abstract machines of the HCL only compute with *closed* HCL terms, that is there are no free variables, the Isabelle terms which correspond to the HCL terms can be terms with free variables. Free variables in the Isabelle term are simply mapped to constants in the HCL term by the administrative layer.

Limited lazy evaluation A major use of congruence rules is to teach the Simplifier that when for example simplifying an *if-then-else* expression, either the *then* or the *else* branch does not need to be simplified if the condition can be simplified to *True* or *False*. The HCL can be taught to exhibit the same behavior. Arguments of a function can be marked to possibly not need evaluation depending on the value of other arguments of that function. The way this marking is done is due to [3]. The HCL uses this feature not only for computing *if-then-else*, but also for evaluating short-circuit logical operators *implies*, *and*, *or* (Fig. 6). For each of these logical operators the second argument is marked as lazy and its evaluation depends on the value of the first argument. For example, if the first argument of *implies* evaluates to *False* then the second argument won't be evaluated. The lazy evaluation feature depends on the abstract machine implementation which may choose to ignore the laziness markings of arguments of a function. All current three abstract machine implementations respect these markings. The Haskell mode does not even need these markings, as it has built-in full lazy evaluation.

Computing in a context Rewrite rules may not only come with conditions, but also with a *context*. A context is also a condition, but one which is not evaluated by the

Name	**Definition**	**Rewrite Rules for HCL**
and	$\mathit{and}\ x\ y = x \wedge y$	$\mathit{and}\ \mathit{True} = \lambda\ y.\ y$ $\mathit{and}\ \mathit{False} = \lambda\ y.\ \mathit{False}$
or	$\mathit{or}\ x\ y = x \vee y$	$\mathit{or}\ \mathit{True} = \lambda\ y.\ \mathit{True}$ $\mathit{or}\ \mathit{False} = \lambda\ y.\ y$
implies	$\mathit{implies}\ x\ y = x \longrightarrow y$	$\mathit{implies}\ \mathit{True} = \lambda\ y.\ y$ $\mathit{implies}\ \mathit{False} = \lambda\ y.\ \mathit{True}$

Fig. 6 Short-circuit Boolean operators of type $\mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool}$

HCL. Look for example at the following theorem which is the externalized form of a theorem encountered in Section 6, Fig. 10 in localized form:

$$\mathit{GraphSystem}\ gs \Longrightarrow \mathit{gs\text{-}Face}\ gs\ d \equiv \mathit{Orbit}\ (\mathit{gs\text{-}face}\ gs)\ d \tag{7}$$

Note that *gs* is a variable, i.e., the above theorem holds for all *gs*. Throughout the computation in which the above rule is applied, *gs* will be fixed but still arbitrary. When the HCL applies the rule, it will keep in mind that it used the condition *GraphSystem gs*. After the computation has finished, the condition is attached as additional assumption to the result. As an example, applying the program that consists only of the above rule to the term

$$\mathit{gs\text{-}Face}\ gs\ d \tag{8}$$

again yields theorem (7).

Limited logical reasoning The most straightforward way of using the HCL is applying a program to an Isabelle term t, yielding a theorem $t \equiv t'$ where t' is the result of evaluating t. Possibly this equation is augmented by contextual assumptions as explained above [29, Sect. 2.6.3]. Often this simple interaction between the HCL and the Isabelle kernel is not enough. Usually the theorem $t \equiv t'$ is not the final theorem you are interested in, but will be used again to produce other theorems. For example, you could use it to discharge the assumption of another theorem, possibly instantiating free variables of this other theorem in the process. After that, you are maybe interested in simplifying the conclusion of the resulting theorem by further evaluating it. When doing this second evaluation, you would like to use the fact that some parts of the term currently being evaluated stem from the instantiation of variables with terms that already have been evaluated. These kinds of scenarios are supported by the HCL that allows to mix modus ponens, delayed substitution and computation without communicating with the Isabelle kernel [29, Sect. 2.6.4].

4.1 Related work

The idea of making computation an integral part of the theorem proving system has been pioneered by the first-order Boyer-Moore theorem prover [6] and its successor ACL2.

The COQ theorem prover provides executable recursively defined functions via the *Fixpoint* command [7, Sect. 1.3.4]. The HCL incorporates the same idea. A key difference is that in COQ defining a function and executing it are inseparably intertwined. Contrarily, the HCL does not care how the functions it executes are defined; it just needs a list of proven conditional equalities.

Berghofer [4] introduced code generation to Isabelle. Haftmann [12] developed the code generation facilities of Isabelle further. Based on these facilities independently of the HCL another method for producing theorems via computation has been developed, called Normalization by Evaluation (NbE) [1]. According to [1] NbE computation is about one order of magnitude slower than generated SML code. This means that NbE computation is also compared with the SML mode of the HCL about one order of magnitude slower. Furthermore, of the 5 main features of the HCL, NbE supports only one, symbolic computation.

5 Tame graphs as hypermaps

For our work we choose to represent planar graphs by *hypermaps*. Hypermaps have been introduced to mechanized theorem proving in [30]. To the attention of the members of the Flyspeck project (particularly Thomas Hales) they came through Gonthier's usage of hypermaps in his proof of the Four Color Theorem [8, 9].

Hypermaps build 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. Our view is basically the one found in [30].

For our purposes only special hypermaps are of interest. An *involutive, planar, connected hypermap* is a tuple (D, e, n, f) where D is called the set of darts and e, n, f are permutations on D such that

$$e = e^{-1} = n \circ f \quad \text{and} \quad |D/e| + |D/f| + |D/n| = |D| + 2 \tag{9}$$

holds. Here D/p denotes the set of equivalence classes induced by the permutation p of D, where two darts $\alpha, \beta \in D$ are considered equivalent if $\alpha = p^k(\beta)$ for some integer k.

Note that all of the tame graphs in our archive fulfill these equations, so we do not need to include them in the definition of the *GraphSystem* predicate in Section 6.

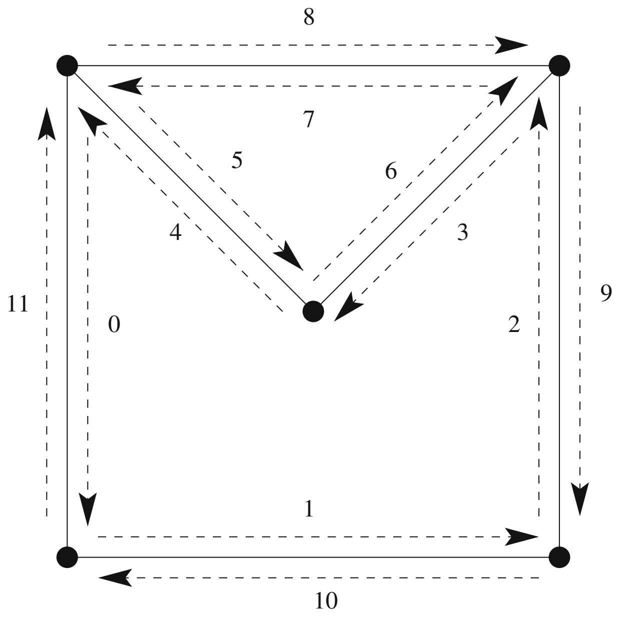

Fig. 7 A connected planar graph (picture reproduced from [28])

As an example, consider the connected planar graph in Fig. 7. Each dart is denoted by a number, and we have $D = \{0, 1, 2, \ldots, 11\}$. The permutations are given by

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

The equivalence classes of f represent the faces of the graph, those of n its nodes, and those of e its edges. The darts themselves can also be 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.

The advantage of representing tame graphs as hypermaps is that the definition of a hypermap is purely combinatorial. Important properties of the tame graph like planarity and orientation are already built into the notion of hypermap, we do not need extra definitions for these. The linear relationships we want to capture later are most naturally expressed on hypermaps. Furthermore, faces, edges and nodes can be represented in a uniform way as finite orbits under the respective permutation. Finite orbits have the nice property that they are sets, but can be traversed just like lists.

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

$$
\begin{aligned}
&\textbf{datatype}\ \omega\ \mathit{NatTreeMap} =\\
&\quad \mathit{TIN}\ \mathit{nat}\ \omega\ (\omega\ \mathit{NatTreeMap})\ (\omega\ \mathit{NatTreeMap})\ |\ \mathit{TNN}
\end{aligned} \qquad (11)
$$

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

$$
\mathit{eval} :: \omega\ \mathit{NatTreeMap} \Rightarrow \mathit{nat} \Rightarrow \omega\ \mathit{option}
$$

$$
\begin{aligned}
&\ \mathit{eval}\ \mathit{TNN}\ x = \mathit{None}\\
&|\ \mathit{eval}\ (\mathit{TIN}\ x\ c\ a\ b)\ x' =\\
&\qquad \mathit{if}\ x{=}x'\ \mathit{then}\ \mathit{Some}\ c\ \mathit{else}\ \mathit{if}\ x' < x\ \mathit{then}\ \mathit{eval}\ a\ x'\ \mathit{else}\ \mathit{eval}\ b\ x' \qquad (12)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{map\text{-}face} :: (\mathit{nat} \times \mathit{nat} \times \mathit{nat})\ \mathit{NatTreeMap} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}\\
&\mathit{map\text{-}face}\ m\ d \equiv \mathit{fst}\ (\mathit{the}\ (\mathit{eval}\ m\ d))\\
&\mathit{map\text{-}edge} :: (\mathit{nat} \times \mathit{nat} \times \mathit{nat})\ \mathit{NatTreeMap} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}\\
&\mathit{map\text{-}edge}\ m\ d \equiv \mathit{fst}\ (\mathit{snd}\ (\mathit{the}\ (\mathit{eval}\ m\ d)))\\
&\mathit{map\text{-}node} :: (\mathit{nat} \times \mathit{nat} \times \mathit{nat})\ \mathit{NatTreeMap} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}\\
&\mathit{map\text{-}node}\ m\ d \equiv \mathit{snd}\ (\mathit{snd}\ (\mathit{the}\ (\mathit{eval}\ m\ d))) \qquad (13)
\end{aligned}
$$

The representation of the permutation functions by binary search trees has been chosen for efficiency because there can be up to about 100 darts in a tame graph and we do frequent lookups. Actually, even this representation proved to be too slow; therefore the HCL is used to calculate all lookups once; each lookup is then represented by its own theorem which is added as rewrite rule. Using the SML mode of the HCL we let Standard ML's pattern matching do the lookup job for us from then on.

The complete archive of 2771 tame graphs is given in our formalization as a constant *Archive* of type ($nat \times nat \times nat$) *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 ($nat \times nat \times nat$) *NatTreeMap*, each constant representing a tame graph.

The conversion from the Flyspeck I format into the binary search tree format has not been formally verified yet. It makes sense to defer such a verification until all other pieces of Flyspeck are there; for example, it could be the case that all other Flyspeck formalizations use hypermaps, in which case it would probably make more sense to rework Flyspeck I in terms of hypermaps instead of verifying a conversion routine.

6 Graph systems

In this section we are giving a definition of the *GraphSystem* predicate introduced in Section 3. The predicate is rather long; nevertheless we will present it here almost in full. The referees found an error in the definition of the predicate in an earlier version of this paper. This error was also present in the actual Isabelle formalization, so all basic linear programs had to be generated and solved again. The result was that now we could only prove about 2% less tame graphs to be contravening! Therefore we think it is important that most of the specification is presented.

We manage the data of a graph system as a record of type ω *GS*. The type parameter ω denotes the type of darts. We require that there is a linear order defined on ω. As in our application ω 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. 8), and those members which represent the real-valued variables on darts (Fig. 9).

Note that although e.g. the type of *gs-σ* is stated to be $\omega \Rightarrow real$, it really is $\omega\ GS \Rightarrow \omega \Rightarrow real$, taking an explicit graph system record as parameter.

gs-darts :: ω set
gs-edge :: $\omega \Rightarrow \omega$
gs-face :: $\omega \Rightarrow \omega$
gs-node :: $\omega \Rightarrow \omega$

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

gs-edges :: ω set
gs-nodes :: ω set
gs-faces :: ω set

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

gs-adjacent :: $\omega \Rightarrow \omega \Rightarrow$ bool
gs-commonquad :: $\omega \Rightarrow \omega \Rightarrow$ bool
gs-separable :: $\omega \Rightarrow$ bool
gs-Separable :: ω set $\Rightarrow \omega \Rightarrow$ bool
gs-tri :: $\omega \Rightarrow$ nat
gs-node-type :: $\omega \Rightarrow$ nat $\Rightarrow$ bool

Fig. 8 Planar graph record components

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

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

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

Fig. 9 Real variable record components

What turns a structure *gs* of type ω *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 [2] fits our purpose. It allows entering a context in which chosen free 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 Figs. 8 and 9 as constants with a built-in implicit *gs* parameter. In the locale context we define e.g.

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

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).

6.1 Topology of a graph system

Let us first make sense of the components shown in Fig. 8. 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 on *darts*.

All other components displayed in Fig. 8 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 Fig. 10 uses the notion of *Orbit*:

$$Orbit\ f\ s = \{g.\ (\exists\, n \in \mathbb{N}.\ g = f^{n}\, s)\} = \{s,\ f\, s,\ f\,(f\, s),\ \ldots\} \tag{14}$$

Fig. 10 Definition of *Face*, *Edge* and *Node*

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

Fig. 11 Definition of *edgerep*, *facerep* and *noderep*

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

Sometimes we need to represent such an equivalence class by a single dart unique to the class. This is why we require the type ω 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. 11). 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. 12).

Finally, the rightmost box of Fig. 8 is made up of further notions which enrich the topology related vocabulary of graph systems. Their definitions are listed in Fig. 13. The Isabelle notation $f\ `\ S$ denotes the image of the set S under the function f, the notation *card* S stands for the cardinality of S if S is finite. Note that we use the short-circuit logical connectives introduced in Fig. 6 to avoid superfluous computation.

6.2 3-space interpretation of a graph system

We already explained in Section 2 how real-valued variables on tame graphs are introduced to model the geometric relationship between a tame graph and its euclidean and spherical graphs.

The real variable components of a graph system in Fig. 9 denote these real-valued variables.

The real variable *yn* α interprets the dart α as a node; hence it must be invariant under the *node* permutation. It denotes the distance that the corresponding center of a ball in the euclidean graph has to the origin. Hence we have $2 \leq \mathit{yn}\, \alpha \leq 2\, t_0$.

The real variable *ye* α interprets α as an edge. It is invariant under the *edge* permutation and measures the length of the corresponding edge in the euclidean graph. Again, we have $2 \leq \mathit{ye}\, \alpha \leq 2\, t_0$.

The real variable *sol* α interprets α as a face. It measures the size of the area of the corresponding surface in the spherical graph. It is invariant under the face permutation. Because the whole surface of a unit ball is $4\,\pi$, we have $0 \leq \mathit{sol}\, \alpha \leq 4\,\pi$.

To explain the real variable *azim* α, let us first interpret α as the arc in the spherical graph that corresponds to the edge α in the tame graph. 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, which is the node in the spherical graph that corresponds to the node α. Now *azim* α measures the spherical angle between those two arcs at p. Because *azim* α is an angle, we have $0 \leq \mathit{azim}\, \alpha \leq 2\,\pi$.

The (in)equalities we have gathered so far are listed in Figs. 14 and 15 as further axioms of a graph system.

Fig. 12 Definition of *edges*, *faces* and *nodes*

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

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

Fig. 13 Topology vocabulary

Additional graph system axioms can be derived by basic geometrical considerations. We know the sum of all angles around a point:

$$\forall N \in nodes.\ 2 * \pi = \sum \alpha \in Node\ N.\ azim\ \alpha \tag{15}$$

So we add this equation to our axioms. 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 another axiom:

$$\forall x \in darts.\ sol\ x = -\ real\ (card\ (Face\ x) - 2) * \pi + \sum \alpha \in Face\ x.\ azim\ \alpha \tag{16}$$

The function *real* converts a value of type *nat* to a value of type *real.*

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 the graph system axiom [15, Eq. 20.1]

$$\forall x \in darts.\ \tau\ x = sol\ x * \zeta * pt - \sigma\ x \tag{17}$$

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,

$$8 * pt \leq \sum \alpha \in faces.\ \sigma\ \alpha \tag{18}$$

must hold. This is the most important of our axioms. From [15] we can assume the bounds shown in Fig. 16 as additional axioms.

The other real variables in the middle and rightmost box in Fig. 9 are special cases of the real variables in the first box; their relationship with the real variables from the first box is given by the axioms in Fig. 17. The significance of these other variables lies in the existence of further axioms which have been converted from a database of inequalities. See [29, Appendix B] for the complete list of these axioms.

Fig. 14 Axioms for invariance under permutation

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

$$\begin{array}{ll}
\forall x \in \mathit{darts}.\ 2 \le \mathit{yn}\ x & \forall x \in \mathit{darts}.\ \mathit{yn}\ x \le 2 * t_0 \\
\forall x \in \mathit{darts}.\ 2 \le \mathit{ye}\ x & \forall x \in \mathit{darts}.\ \mathit{ye}\ x \le 2 * t_0 \\
\forall x \in \mathit{darts}.\ 0 \le \mathit{azim}\ x & \forall x \in \mathit{darts}.\ \mathit{azim}\ x \le 2 * \pi \\
\forall x \in \mathit{darts}.\ 0 \le \mathit{sol}\ x & \forall x \in \mathit{darts}.\ \mathit{sol}\ x \le 4 * \pi
\end{array}$$

Fig. 15 Axioms for basic geometrical bounds

6.3 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 a formalization of [13, Group 4, rule 1]:

$$\forall \alpha \in \mathit{nodes}.\ \mathit{node\text{-}type}\ \alpha\ 4\ \mathit{implies}\ \left(\sum \beta \in \mathit{Node}\ \alpha .\ \sigma\ \beta \le 33\ /\ 100 * \mathit{pt}\right) \tag{19}$$

Second, here is an axiom that is a formalization of [13, Group 4, rule 3]:

$$\forall \alpha \in \mathit{nodes}.\ \mathit{node\text{-}type}\ \alpha\ 5\ \mathit{implies}$$
$$\left(\sum \beta \in \mathit{Node}\ \alpha .\ \sigma\ \beta + 419351\ /\ 1000000 * \mathit{sol}\ \beta - 2856354\ /\ 10000000\right) \le 0 \tag{20}$$

Finally, Figs. 18 and 19 complete the set of axioms. The functions *const-a* and *const-d* appearing in Fig. 19 map natural numbers to real numbers and correspond to the functions a and b defined in [15, Sect. 18.2]. While all other axioms are more or less a direct translation of the corresponding statements in [15], the axioms in these last two figures need a little bit of explanation.

Lemma 10.6 in [15] demands $v_1, \ldots, v_k$ to be distinct vertices for some $k \le 4$ and then states inequalities in terms of these vertices. In Fig. 18 we translate this by treating each of the cases $k = 1, 2, 3, 4$ separately; for example, the first axiom treats the case $k = 4$ and the vertices v_1, v_2, v_3, v_4 are denoted by the darts $\alpha, \beta, \gamma, \delta$.

Lemma 22.12 in [15] states an inequality for any separated set V of vertices. From the definition of separated set [15, Sect. 18.2] we checked that the archive of tame

$$\begin{array}{l}
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 3\ \mathit{implies}\ 0 \le \tau\ x \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 4\ \mathit{implies}\ 1317\ /\ 10000 \le \tau\ x \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 5\ \mathit{implies}\ 27113\ /\ 100000 \le \tau\ x \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 6\ \mathit{implies}\ 41056\ /\ 100000 \le \tau\ x \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 7\ \mathit{implies}\ 54999\ /\ 100000 \le \tau\ x \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 8\ \mathit{implies}\ 6045\ /\ 10000 \le \tau\ x \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 3\ \mathit{implies}\ \sigma\ x \le \mathit{pt} \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 4\ \mathit{implies}\ \sigma\ x \le 0 \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 5\ \mathit{implies}\ \sigma\ x \le -5704\ /\ 100000 \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 6\ \mathit{implies}\ \sigma\ x \le -11408\ /\ 100000 \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 7\ \mathit{implies}\ \sigma\ x \le -17112\ /\ 100000 \\
\forall x \in \mathit{darts}.\ \mathit{card}\ (\mathit{Face}\ x) = 8\ \mathit{implies}\ \sigma\ x \le -22816\ /\ 100000
\end{array}$$

Fig. 16 Bounds for σ and τ from [15, Lemma 20.2]

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

Fig. 17 Variations of real variables

$$\begin{aligned}
&\forall \alpha \in \textit{nodes}.\ \textit{node-type}\ \alpha\ 5\ \textit{implies}\\
&\quad (\forall \beta \in \textit{nodes}.\ \alpha < \beta\ \textit{and node-type}\ \beta\ 5\ \textit{implies}\\
&\quad (\forall \gamma \in \textit{nodes}.\ \beta < \gamma\ \textit{and node-type}\ \gamma\ 5\ \textit{implies}\\
&\quad (\forall \delta \in \textit{nodes}.\ \gamma < \delta\ \textit{and node-type}\ \delta\ 5\ \textit{implies}\\
&\quad 55\ /\ 100 * 4 * \textit{pt} \le (\textstyle\sum\ S \in \textit{faces} \cap \textit{facerep}\ `\ (\textit{Node}\ \alpha \cup \textit{Node}\ \beta \cup \textit{Node}\ \gamma \cup \textit{Node}\ \delta).\ \tau\ S)\\
&\quad \wedge (\textstyle\sum S \in \textit{faces} \cap \textit{facerep}\ `\ (\textit{Node}\ \alpha \cup \textit{Node}\ \beta \cup \textit{Node}\ \gamma \cup \textit{Node}\ \delta).\ \sigma\ S - \textit{pt}) \le -\ 48\ /\ 100 * 4 * \textit{pt})))\\
&\forall \alpha \in \textit{nodes}.\ \textit{node-type}\ \alpha\ 5\ \textit{implies}\\
&\quad (\forall \beta \in \textit{nodes}.\ \alpha < \beta\ \textit{and node-type}\ \beta\ 5\ \textit{implies}\\
&\quad (\forall \gamma \in \textit{nodes}.\ \beta < \gamma\ \textit{and node-type}\ \gamma\ 5\ \textit{implies}\\
&\quad 55\ /\ 100 * 3 * \textit{pt} \le (\textstyle\sum\ S \in \textit{faces} \cap \textit{facerep}\ `\ (\textit{Node}\ \alpha \cup \textit{Node}\ \beta \cup \textit{Node}\ \gamma).\ \tau\ S)\\
&\quad \wedge (\textstyle\sum S \in \textit{faces} \cap \textit{facerep}\ `\ (\textit{Node}\ \alpha \cup \textit{Node}\ \beta \cup \textit{Node}\ \gamma).\ \sigma\ S - \textit{pt}) \le -\ 48\ /\ 100 * 3 * \textit{pt}))\\
&\forall \alpha \in \textit{nodes}.\ \textit{node-type}\ \alpha\ 5\ \textit{implies}\\
&\quad (\forall \beta \in \textit{nodes}.\ \alpha < \beta\ \textit{and node-type}\ \beta\ 5\ \textit{implies}\\
&\quad 55\ /\ 100 * 2 * \textit{pt} \le (\textstyle\sum\ S \in \textit{faces} \cap \textit{facerep}\ `\ (\textit{Node}\ \alpha \cup \textit{Node}\ \beta).\ \tau\ S)\\
&\quad \wedge (\textstyle\sum S \in \textit{faces} \cap \textit{facerep}\ `\ (\textit{Node}\ \alpha \cup \textit{Node}\ \beta).\ \sigma\ S - \textit{pt}) \le -\ 48\ /\ 100 * 2 * \textit{pt})\\
&\forall \alpha \in \textit{nodes}.\ \textit{node-type}\ \alpha\ 5\ \textit{implies}\\
&\quad 55\ /\ 100 * 1 * \textit{pt} \le (\textstyle\sum\ S \in \textit{faces} \cap \textit{facerep}\ `\ \textit{Node}\ \alpha.\ \tau\ S)\\
&\quad \wedge (\textstyle\sum S \in \textit{faces} \cap \textit{facerep}\ `\ \textit{Node}\ \alpha.\ \sigma\ S - \textit{pt}) \le -\ 48\ /\ 100 * 1 * \textit{pt}
\end{aligned}$$

Fig. 18 Axioms from [15, Lemma 10.6]

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

Fig. 19 Axioms from [15, Lemma 22.12]

graphs contains no separated set V with more than three vertices. A case split on $|V|$ yields the 3 axioms in Fig. 19; for example, the last axiom treats the case $|V| = 3$ and is formulated in terms of v_1, v_2, v_3, where $V = \{v_1, v_2, v_3\}$.

Thus we have defined the predicate *GraphSystem* of type ω *GS* $\Rightarrow$ *bool* which is true for some *gs* iff *gs* fulfills all the axioms and definitions listed in Section 6 (including those further axioms from the database of inequalities).

7 Generating and running the basic linear programs

In this section we describe how we prove formally within Isabelle/HOL that a given tame graph is not a graph system.

7.1 Linking tame graph and graph system

For each tame graph, we assume that it fulfills all graph system axioms. In order to express this assumption, we need to connect the abstract notion of a graph system with our concrete representation of tame graphs. We do this via the relation *PGS*.

$$\begin{aligned}
&\textit{func-eq} :: \omega\ \textit{GS} \Rightarrow (\omega \Rightarrow \omega') \Rightarrow (\omega \Rightarrow \omega') \Rightarrow \textit{bool}\\
&\textit{func-eq gs f g} = (\forall\ d.\ d \in \textit{gs-darts gs} \longrightarrow f\ d = g\ d)
\end{aligned}$$

$$\begin{aligned}
&\textit{PGS} :: \textit{nat GS} \Rightarrow (\textit{nat} \times \textit{nat} \times \textit{nat})\ \textit{NatTreeMap} \Rightarrow \textit{bool}\\
&\textit{PGS gs S} = (\textit{GraphSystem gs}\\
&\qquad \wedge\ \textit{gs-darts gs} = \textit{dom}\ (\textit{eval S})\\
&\qquad \wedge\ \textit{func-eq gs}\ (\textit{gs-face gs})\ (\textit{map-face S})\\
&\qquad \wedge\ \textit{func-eq gs}\ (\textit{gs-edge gs})\ (\textit{map-edge S})\\
&\qquad \wedge\ \textit{func-eq gs}\ (\textit{gs-node gs})\ (\textit{map-node S}))
\end{aligned} \tag{21}$$

The *PGS* relation is necessary because we do not want to describe as part of the graph system notion how to represent the permutation functions and the set of darts; we want to keep the notion of graph system as abstract as possible. Currently we use the binary search trees from Section 5 to represent permutation functions and darts; but this may change at a later stage of the Flyspeck project.

For a given tame graph S, say $S = \textit{graph-1}$, we can then enter the context *PGS gs S*. The HCL (Section 4) allows us to perform computations within this context. If the HCL did not have that capability, it would have been difficult, maybe even impossible, to apply the HCL to our problem, because all of our computations need to be done in a context which provides the underlying implicit assumption *PGS gs S*. Our goal is to prove *False* in this context, i.e., to show *PGS gs S* $\Longrightarrow$ *False*.

Instead of defining a relation *PGS*, one could try to circumvent having to compute within a context by defining a function *Convert* which maps a tame graph of type (*nat* $\times$ *nat* $\times$ *nat*) *NatTreeMap* to a graph system record of type *nat GS*. Then the goal would be to show *GraphSystem (Convert S)* = *False*. One problem with this approach is that the expression *GraphSystem (Convert S)* expands to a big confusing term that is hard to deal with. More importantly, it has basically the following structure:

$$E_1 \wedge E_2 \wedge \ldots \wedge E_n \wedge U \tag{22}$$

The E_i are ∀-quantified equations that we would like to use to rewrite U. Also we would like to hand this rewriting job over to the HCL. But we cannot use the E_i directly as an HCL program because the HCL accepts only theorems, and the E_i are mere terms. We could work around this problem by creating a program out of the theorems $E_i \Longrightarrow E_i$, effectively introducing a context consisting of n components. It seems easier to avoid doing this and work from the start within the context $PGS\ gs\ S$, i.e., to use theorems $PGS\ gs\ S \Longrightarrow E_i'$ to rewrite U'.

The first theorems we compute with the HCL enable us to abstract from the concrete tame graph representation. We demand that the set of darts is represented as the *cyclic* orbit of an executable function. We call $Orbit\ f\ s$ cyclic if $s \in Orbit\ f\ (fs)$. For the tame graphs from the archive the orbit function is defined via

$$natsection\text{-}gen\ n\ x = \mathit{if}\ n \leq x + 1\ \mathit{then}\ 0\ \mathit{else}\ x + 1 \tag{23}$$

For each tame graph S in the archive we prove a theorem of the form

$$PGS\ gs\ S \Longrightarrow gs\text{-}darts\ gs = Orbit(natsection\text{-}gen\ m)0 \tag{24}$$

where m depends on S. For example, for $S = graph\text{-}1$ we have $m = 48$. Then we prove for each dart $d \in gs\text{-}darts\ gs$ the theorems

$$\begin{aligned} &PGS\ gs\ S \Longrightarrow gs\text{-}face\ gs\ d = d_f \\ &PGS\ gs\ S \Longrightarrow gs\text{-}node\ gs\ d = d_n \\ &PGS\ gs\ S \Longrightarrow gs\text{-}edge\ gs\ d = d_e \end{aligned} \tag{25}$$

where the d_f, d_n and d_e obviously depend on S and d.

Computing first all needed theorems about the permutation functions and the set of darts is not only useful for abstraction; that way, instead of computing the permutation functions over and over again, we can convert the theorems into an HCL program, effectively *caching* these computations. Evaluating a permutation function is now done via a simple lookup. Because we use the SML mode of the HCL, the lookup is implemented by compiled pattern matching code.

7.2 Folding the orbit

We have represented the set of darts, and the face, edge and node equivalence classes as cyclic orbits. Many other objects appearing in the definition of a graph system are obtained by operating on these cyclic orbits. Therefore it is important that we have a convenient way of operating on and computing with cyclic orbits. Whether an orbit is cyclic is stated via the *orbit-terminates* predicate:

$$orbit\text{-}terminates\ f\ x\ x' = x \in Orbit\ f\ (f\ x') \tag{26}$$

$Orbit\ f\ s$ is cyclic iff $orbit\text{-}terminates\ f\ s\ s$ holds. If an orbit is cyclic, then we can prove this by computing with the recursion equation

$$\begin{aligned} &orbit\text{-}terminates\ f\ x\ x' = \\ &\quad \mathit{if}\ f\ x' \neq x\ \mathit{then}\ orbit\text{-}terminates\ f\ x\ (f\ x')\ \mathit{else}\ True \end{aligned} \tag{27}$$

A cyclic orbit is a set; nevertheless, computationally it is similar to a list. In particular, we can canonically iterate through all elements of a cyclic orbit. Sets are nice for reasoning; lists are nice for computing. For our application, cyclic orbits combine these properties of sets and lists advantageously. We exploit this via the *fold-section* functional which mimics folding over a list. It is defined by

$$\textit{fold-section continue } h\ \ f\ g\ z\ a = \textit{For continue } h\ (\lambda\ z.\ \lambda\ a.\ \ f\ (g\ z)\ a)\ z\ a \tag{28}$$

In above definition we use the *For* combinator. This combinator allows to define partial functions which are basically loops. The following loop in the programming language C

```
for (i = i0, a = a0; continue(i); i = f(i))
    a = A(i, a);
a1 = a;
```

can be modelled in HOL by the equation

$$a_1 = \textit{For}\ \ \textit{continue}\ \ \ f\ A\ i_0\ a_0. \tag{29}$$

The *For* combinator is defined and explained in more detail in [28].

Computing with *fold-section* is done using the recursion equation

$$\begin{aligned}&\textit{fold-section continue } h\ \ f\ g\ x\ ac = \\ &\quad \textit{if continue } x \textit{ then fold-section continue } h\ \ f\ g\ (h\ x)\ (f\ (g\ x)\ ac)\ \textit{else } ac\end{aligned} \tag{30}$$

Reasoning about *fold-section* on cyclic orbits is facilitated by connecting it with the fold functional on finite sets which is described in [24]. The following theorem establishes this connection:

$$\begin{aligned}&\textit{ACf } f \Longrightarrow \textit{orbit-terminates } h\ z\ z \Longrightarrow \\ &\quad \textit{fold } f\ g\ a\ (\textit{Orbit } h\ z) = \textit{fold-section } (\lambda\ y.\ y \neq z)\ h\ \ f\ g\ (h\ z)\ (f\ (g\ z)\ a)\end{aligned} \tag{31}$$

The condition *ACf* f expresses that because we are now talking about arbitrary finite sets, which impose no order on their elements, we need the binary operation f to be associative and commutative.

This gives us all the tools we need to define functions over cyclic orbits, and to reason about and compute with these functions. For example, we define

$$\begin{aligned}\textit{sum-orbit}\ \ f\ g\ x\ x'\ a\ &=\ \textit{fold-section}\ (\lambda\ y.\ y \neq x)\ \ f\ (\lambda\ u\ v.\ u + v)\ g\ (f\ x')\ a \\ \textit{or-orbit}\ \ f\ g\ x\ x'\ a\ &=\ \textit{fold-section}\ (\lambda\ y.\ y \neq x)\ \ f\ (\lambda\ p\ q.\ p \vee q)\ g\ (f\ x')\ a \\ \textit{and-orbit}\ \ f\ g\ x\ x'\ a\ &=\ \textit{fold-section}\ (\lambda\ y.\ y \neq x)\ \ f\ (\lambda\ p\ q.\ p \wedge q)\ g\ (f\ x')\ a\end{aligned} \tag{32}$$

Using Isabelle's reasoning machinery for finite sets it is easy to prove

$$\begin{aligned}&\textit{orbit-terminates}\ \ f\ x\ x \Longrightarrow (\forall\ y \in \textit{Orbit}\ \ f\ x.\ \ P\ y) = \textit{and-orbit}\ \ f\ \ P\ x\ x\ (P\ x) \\ &\textit{orbit-terminates}\ \ f\ x\ x \Longrightarrow (\exists\ y \in \textit{Orbit}\ \ f\ x.\ \ P\ y) = \textit{or-orbit}\ \ f\ \ P\ x\ x\ (P\ x) \\ &\textit{orbit-terminates}\ \ f\ x\ x \Longrightarrow \left(\sum\ y \in \textit{Orbit}\ \ f\ x.\ q\ y\right) = \textit{sum-orbit}\ \ f\ q\ x\ x\ (q\ x) \\ &\textit{orbit-terminates}\ \ f\ x\ x \Longrightarrow \textit{card}\ (\textit{Orbit}\ \ f\ x) = \textit{sum-orbit}\ \ f(\lambda\ y.\ 1)\ x\ x\ 1\end{aligned} \tag{33}$$

We can give all of these theorems together with a bunch of similar theorems to the HCL and run the resulting program on those graph system axioms that involve real variables.

In many axioms we used instead of the normal implication operator $\longrightarrow$ the short-circuit operator *implies* from Fig. 6. This is essential in order to be able to execute certain axioms like those in Figs. 18 or 19, which contain nested implications, in reasonable time, because if the precondition of the implication evaluates to *False*, which is much more often the case than not, then we don't bother evaluating the conclusion of the implication.

The result of using the HCL to normalize an instance of the graph system axioms for a particular tame graph is a large conjunction of inequalities in the real variables. We can further normalize this conjunction, again using the HCL, to yield the theorem

$$PGS\ gs\ S \quad \Longrightarrow \quad A * x \leq b \tag{34}$$

where A, b and x are finite matrices. The elements of the matrix A and the vector b are terms denoting constant expressions like $(4 * \pi/\sqrt{2})$, the elements of the vector x are the real variables of the graph system.

7.3 Proving infeasibility with finite matrices

In this section we abridge material which can be found in much greater detail in [29, Chapt. 3].

7.3.1 Finite matrices

Finite matrices have been introduced in [27]. Obviously, matrices should form a type in HOL, but how does one deal with the dimension of a matrix? There are no *dependent types* in HOL, so it seems impossible to have a parametrized family of types where the dimension of the matrix would be the parameter. But it is possible; one possibility that is pursued by Harrison [17] in his Hol-light system is to represent the needed parameter by type variables! In our case this idea would cause serious problems later on when we work with concrete matrix representations.

Our approach exploits the fact that the matrix elements commonly used in mathematics [21] carry an algebraic structure, that of a ring, which always contains a zero. We define the type of *finite matrices* of elements of type ω to be ω *matrix*, where ω must contain a zero:

$$\begin{aligned}&\textbf{type}\ \omega\ \mathit{infmatrix} = \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \omega\\ &\textbf{typedef}\ \omega\ \mathit{matrix} = \{f :: (\omega :: \mathit{zero})\mathit{infmatrix} \mid \mathit{finite}\{(j,i) \mid f\ j\ i \neq 0\}\}\ .\end{aligned} \tag{35}$$

Therefore the type ω *matrix* can intuitively be understood to consist of those infinite matrices that have finite support. Any finite matrix has a minimal dimension $N \times M$ which depends on its support. But it can be used as a matrix of any dimension $n \times m$ with $n \geq N$, $m \geq M$. For example, adding two finite matrices of different minimal dimensions is no problem:

$$\begin{pmatrix}1&7&3\\2&1&0\end{pmatrix} + \begin{pmatrix}0\\2\\10\\5\end{pmatrix} = \begin{pmatrix}1&7&3\\2&1&0\\0&0&0\\0&0&0\end{pmatrix} + \begin{pmatrix}0&0&0\\2&0&0\\10&0&0\\5&0&0\end{pmatrix} = \begin{pmatrix}1&7&3\\4&1&0\\10&0&0\\5&0&0\end{pmatrix}$$

The first operand of above sum has minimal dimension 2×3, the second operand has minimal dimension 4×1, and the resulting finite matrix has minimal dimension 4×3.

Birkhoff points out [5, Chapt. 17] that for a fixed n the ring of all $n \times n$ matrices forms a lattice-ordered ring in a natural way if ω is one, too. The same is true for our finite matrices! We used this fact extensively when reasoning about finite matrices in Isabelle/HOL.

Note that we do not introduce special types for column or row vectors. A column vector is just a finite matrix of minimal column dimension $m \leq 1$, a row vector is a finite matrix of minimal row dimension $n \leq 1$.

7.3.2 Representing finite matrices

We are interested in calculating with concrete matrices of type *real matrix*. Let us first explain how we represent concrete reals. One possibility would have been to represent real numbers r by fractions $r = a/b$ where a and b are integers. The problem with fractions is that addition is an expensive operation. Instead, we chose to represent reals as binary, arbitrary precision floating point numbers $\mathit{float}(m, e)$ instead:

$$\mathit{float}\ (m, e) = (\mathit{real}\ m) * 2^e \tag{36}$$

In the above, the integer m is called the mantissa, and the integer e the exponent. Note that (real m) denotes the conversion of the integer m to a real number. Floats do not form a new type; they are just notation for certain real numbers.

A disadvantage of floating point numbers is that some fractions like $1/3$ cannot be represented by them; but this is not a problem, since we have to work with approximations for real numbers like π anyway. Therefore, when doing computations, we actually represent real numbers by intervals $[f_1, f_2]$ of floats with $f_1 \leq f_2$.

We have formalized executable basic arithmetical operations like addition, multiplication and division for intervals of floats. Recently there has been work to extend the set of available executable functions on floats by those functions which have Taylor approximations [18]. During Flyspeck II, these extensions have not been available yet. Therefore we do not introduce π and other constants like $\sqrt{2}$ by their usual mathematical definitions, but introduce unspecified symbols for them and state bounds for them axiomatically. It should be easy now to remedy this.

Let us now explain how we represent matrices. We use sparse matrices for this which we represent by lists; a sparse matrix is a sparse vector of sparse vectors of reals; a sparse vector of elements of type ω is a list of pairs (i, w) where i is called the index, and w is of type ω. The *sparse-row-matrix* morphism translates sparse matrices to finite matrices like in the following example:

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

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

The *sparse-row-matrix* morphism interprets the inner lists as the rows of the matrix. It does not assume any sortedness constraints on sparse matrices, not even uniqueness of indices; but the operations we have formalized for sparse matrices do assume such constraints.

Basic executable operations like addition and multiplication have been formalized for sparse matrices, following algorithms given in [11], and proven to be correct with respect to their finite matrix counter parts and the *sparse-row-matrix* morphism, assuming certain sortedness constraints. This is 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.

7.3.3 Proving infeasibility

We need to prove

$$A * x \leq b \quad \Longrightarrow \quad \text{False} \tag{37}$$

Just like we need intervals to approximate real numbers, we make also use of intervals to approximate matrices with reals as elements. The matrix A is approximated by two matrices A_1 and A_2 that have only floats as entries and that bound A such that $A_1 \leq A \leq A_2$ holds. The matrix b is approximated by a matrix b_0 such that $b \leq b_0$.

Our strategy is to solve a linear program to prove above implication. We want to use an external linear programming tool for this; the approximate solution of the linear program will serve us as a certificate. It seems that in order to implement this strategy systematically, we need to find matrices x_1 and x_2 consisting only of floating point entries such that $x_1 \leq x \leq x_2$ holds. This is necessary because the certificate introduces certain rounding errors; in order to bound these errors, we need to bound x. Fortunately the structure of A always allows us to calculate such a-priori bounds for x easily [29, Sect. 3.8]. Again, we calculate a certificate for these a-priori bounds outside Isabelle/HOL, and then check it cheaply within the Isabelle/HOL logic. The only alternative to needing the bounds for x seems to be to formalize a linear programming algorithm within Isabelle/HOL. Even using the HCL, being a purely functional algorithm this would probably be considerably slower than our current approach; in order to be remotely competitive, the HCL would also need to incorporate native floating point support, which it currently does not.

We choose a matrix A_0 consisting only of floating point numbers such that $A_1 \leq A_0 \leq A_2$, for example $A_0 = A_1$. Furthermore we choose a float $K < 0$, for example $K = -1$. In theory, it does not matter which $K < 0$ we choose, but in practice a K too close to 0 is dangerous; for numerical reasons $K \leq -1$ is sufficient for our purposes. We then define the linear program L as

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

where t is a new variable we introduce and

$$c' = \begin{pmatrix} 0 & K \end{pmatrix}, \quad x' = \begin{pmatrix} x \\ t \end{pmatrix}, \quad A' = \begin{pmatrix} A_0 & b_0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix}, \quad b' = \begin{pmatrix} b_0 \\ 1 \\ 0 \end{pmatrix}.$$

The linear program L is feasible and bounded [29, Sect. 3.5] and we can therefore solve its dual L' by an external linear program solver, in our case the GNU linear programming kit. The dual L' is defined as

$$\text{Minimize } y' * b' \text{ where } y' \text{ is subject to the conditions } y' * A' = c' \text{ and } y' \geq 0.$$

This gives us an approximate solution $y' = \big(y\ y_1\ y_2\big) \geq 0$ of L'. Here y is a row vector, and y_1 and y_2 are floats. Plugging y and y_1 into the Isabelle theorem

$$\begin{aligned} & A * x \leq b \Longrightarrow A_1 \leq A \Longrightarrow A \leq A_2 \Longrightarrow b \leq b_0 \Longrightarrow x_1 \leq x \Longrightarrow x \leq x_2 \\ & \Longrightarrow 0 \leq y \Longrightarrow 0 \leq y_1 \Longrightarrow \\ & \Longrightarrow 0 \leq y * b_0 + y_1 + \big(\text{let } s_1 = -y * A_2; s_2 = -y * A_1 \\ & \qquad\qquad\qquad \text{in } s_2^+ * x_2^+ + s_1^+ * x_2^- + s_2^- * x_1^+ + s_1^- * x_1^-\big), \end{aligned} \tag{38}$$

discharging all assumptions except $A * x \leq b$ and computing the conclusion proves (37) if (37) is true. If (37) is false we obtain the not very useful theorem $A * x \leq b \Longrightarrow 0 \leq C$ for some non-negative float C.

In (38) the notations q^+ and q^- stand for the positive and the negative part of q, where q belongs to some lattice-ordered ring. In our case, it is the ring of finite matrices over the real numbers; the positive part of a finite matrix is the matrix you get by replacing all negative entries by 0, the negative part is obtained by replacing all positive entries of the matrix with 0.

The proof of (38) is via standard algebraic manipulations in lattice-ordered rings.

8 Results

We were able to prove the inconsistency of 2504 of the graph systems. This yields a success rate of $2504/(2771 - 2) \approx 90.4\%$.

The complete Flyspeck II formalization can be downloaded from [26].

We used the SML mode of the HOL Computing Library for all larger computations within the theorem prover Isabelle. The computing resources needed were still enormous. Therefore we were forced to examine each tame graph by its own Isabelle process. In the original verification each Isabelle process ran on a dedicated processor of a cluster of 32 four processor 2.4 GHz Opteron 850 machines with 8 GB RAM per machine. The quickest process needed 8.4 min, the slowest 67. The examination of *all* tame graphs took about 7.5 h of cluster runtime. This corresponds to about 40 days on a single processor machine.

Note that most of the computing time was spent generating the basic linear program, approximating it, and calculating the a-priori bounds. The actual time needed by the external linear program solver *GLPK*(GNU linear programming kit) was only a matter of seconds for each tame graph. A considerable performance speed-up should be experienced if caching were added to the HCL.

How reliable are our results? The major source of potential mayhem is that some mistake might have been introduced in the specification of the basic linear programs. Indeed, the referees discovered an error in the definition of the *adjacent* relation in an earlier version of this article. There, instead of the definition now given in Fig. 13, we used the wrong definition

$$\forall\ \alpha\ \beta.\ \mathit{adjacent}\,\alpha\ \beta = (\exists\, n \in \mathit{Node}\,\alpha.\ \mathit{edge}\,\alpha \in \mathit{Node}\,\beta) \tag{39}$$

Another error, which was discovered by Hales while reviewing the PhD thesis of the first author, was due to ambiguous phrasing in [15]. Instead of axiom (19) we wrongly used

$$\forall \alpha \in \mathit{nodes}.\ \mathit{node\text{-}type}\,\alpha\ 4\ \mathit{implies}\,\forall \beta \in \mathit{Node}\,\alpha.\ \sigma\ \beta \leq 33\ /\ 100 * \mathit{pt} \tag{40}$$

Considering the length of the specification of a graph system, other errors might have crept in and still hide in the formalization. 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. We can console ourselves that the methods presented here are general enough so that a transfer to a corrected specification should always be possible, and probably easy so.

Another potential source of mistakes is the use of the HOL Computing Library. After all, it is just a piece of unverified software which has been tested by only a very small number of persons, and there is no mechanized proof yet for its correctness.

So what about the remaining roughly 10% of tame graphs which still need to be shown to be non-contravening? For them it is necessary to look at a more complicated definition of what a graph system is. Remember, a graph system is a linear approximation of the non-linear geometrical constraints imposed on contravening tame graphs. For the remaining 10% of tame graphs this approximation is just not good enough. This corresponds to what Thomas Hales did in his 1998 proof; he also first arrived at the elimination of about 90% of tame graphs and then needed to study the rest in more detail. To do so in Isabelle, too, one needs to expand the definition of a graph system and the linear program generation process such that one tame graph induces a series of linear programs, not only one basic linear program. This is future work.

How much work has it been so far? From start to end it took about four years, but not all of this time was exclusively devoted to the work presented in this paper. So it is hard to say how much work it really is. Maybe the more interesting question is how much time it *should* have taken given the right theorem proving infrastructure. A lot of time was devoted to formalize things "the right way". For example, finding the finite matrices abstraction to talk about linear programming. Or then, after having this abstraction, using it not only for reasoning, but also for computing. Another example is using hypermaps for representing planar graphs; instead of using a list based representation, hypermaps automatically suggested the development of the orbit notion within Isabelle. Using these abstractions to the extent that we have done would not have been possible without the HCL, which makes computing with them efficient.

So, from this point of view, the right theorem proving infrastructure would already include something like the HCL, there would already be formalizations of orbits, finite matrices and how to prove bounds of linear programs; it would also include interval arithmetic on floating point numbers, and executable Taylor approximations of functions like *sine* and *cosine*. These are topics that are very likely interesting not only to Flyspeck II, but to many other formalizations of mathematics and mathematical algorithms. Given this infrastructure, Flyspeck II could have been completed within a month.

The ultimate goal is to make theorem proving technology so accessible to mathematicians that Hales would have done his work from the start in a mechanized and formal setting. Flyspeck II would have been superfluous, then.

Acknowledgements Many thanks to Thomas Hales for his continuing support during this project; without it its success would not have been possible. Also many thanks to the anonymous referees; their diligence saved the day.

References

1. Aehlig, K., Haftmann, F., Nipkow, T.: A compiled implementation of normalization by evaluation. In: Mohamed, A., Munoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 39–54. Springer, New York (2008)
2. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES. Lecture Notes in Computer Science, vol. 3085, pp. 34–50. Springer, New York (2003)
3. Barras, B.: Programming and computing in HOL. In: Aagaard, M., Harrison, J. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 1869, pp. 17–37. Springer, New York (2000)
4. Berghofer, S.: Proofs, programs and executable specifications in higher order logic. Ph.D. thesis, Technische Universität München (2003)
5. Birkhoff, G.: Lattice Theory. AMS, Providence (1967)
6. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic, New York (1979)
7. The COQ development team: The COQ reference manual, version 8.2. http://coq.inria.fr (2009)
8. Gonthier, G.: A computer-checked proof of the four color theorem. http://research.microsoft.com/~gonthier/4colproof.pdf
9. Gonthier, G.: Formal proof—the four color theorem. Not. Am. Math. Soc. **55** (2008)
10. Gordon, M.: From LCF to HOL: a short history. In: Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 169–185. MIT, Cambridge (2000)
11. Gustavson, F.G.: Two fast algorithms for sparse matrices: multiplication and permuted transposition. ACM Trans. Math. Softw. **4**(3), 250–269 (1978)
12. Haftmann, F.: Code generation from specifications in higher-order logic. Ph.D. thesis, Technische Universität München (2009)
13. Hales, T.C.: Sphere packings III. arXiv:math/9811075v2
14. Hales, T.C., Ferguson, S.P.: A proof of the Kepler conjecture. Ann. Math. **162**, 1065–1185 (2005)
15. Hales, T.C., Ferguson, S.P.: The Kepler conjecture. Discrete Comput. Geom. **36**, (2006)
16. Hales, T.C., Harrison, J., McLaughlin, S., Nipkow, T., Obua, S., Zumkeller, R.: A revision of the proof of the Kepler Conjecture. Discrete Comput. Geom. (2009)
17. Harrison, J.: A HOL theory of Euclidean space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. Lecture Notes in Computer Science, vol. 3603. Springer, Oxford (2005)
18. Hölzl, J.: Proving inequalities over reals with computation in Isabelle/HOL. In: Proceedings of the ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009) (2009)
19. Hurd, J., Melham, T.F. (eds.): Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, 22–25 August 2005, Proceedings. Lecture Notes in Computer Science, vol. 3603. Springer, New York (2005)
20. Krauss, A.: Partial recursive functions in higher-order logic. In: Furbach, U., Shankar, N. (eds.) IJCAR. Lecture Notes in Computer Science, vol. 4130, pp. 589–603. Springer, New York (2006)
21. Lang, S.: Algebra. Addison-Wesley, Reading (1974)
22. Nipkow, T., Bauer, G., Schultz, P.: The archive of tame graphs. http://www4.informatik.tu-muenchen.de/~nipkow/pubs/Flyspeck (2006)
23. Nipkow, T., Bauer, G., Schultz, P.: Flyspeck I: tame graphs. In: IJCAR, pp. 21–35 (2006)
24. Nipkow, T., Paulson, L.C.: Proof pearl: defining functions over finite sets. In: Hurd, J., Melham, T.F. (eds.) Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, 22–25 August 2005, Proceedings. Lecture Notes in Computer Science, vol. 3603, pp. 385–396. Springer, New York (2005)
25. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—a proof assistant for higher-order logic. In: Lecture Notes in Computer Science, vol. 2283. Springer, New York (2002)
26. Obua, S.: Flyspeck II: the basic linear programs. http://www4.in.tum.de/~obua/flyspeckII (2007)
27. Obua, S.: Proving bounds for real linear programs in Isabelle/Hol. In: Hurd, J., Melham, T.F. (eds.) Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, 22–25 August 2005, Proceedings. Lecture Notes in Computer Science, vol. 3603, pp. 227–244. Springer, New York (2005)

28. Obua, S.: Proof pearl: looping around the orbit. In: Schneider, K., Brandt, J. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 4732, pp. 223–231. Springer, New York (2007)
29. Obua, S.: Flyspeck II: the basic linear programs. Ph.D. thesis, Technische Universität München (2008)
30. Puitg, F., Dufourd, J.-F.: Formal specification and theorem proving breakthroughs in geometric modeling. In: Grundy, J., Newey, M.C. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 1479, pp. 401–422. Springer, New York (1998)