Verified Efficient Enumeration of Plane Graphs Modulo Isomorphism

Tobias Nipkow

Institut für Informatik, Technische Universität München

Abstract. Due to a recent revision of Hales's proof of the Kepler Conjecture, the existing verification of the central graph enumeration procedure had to be revised because it now has to cope with more than 10^9 graphs. This resulted in a new and modular design. This paper primarily describes the reusable components of the new design: a while combinator for partial functions, a theory of worklist algorithms, a stepwise implementation of a data type of sets over a quasi-order with the help of tries, and a plane graph isomorphism checker. The verification turned out not to be in vain as it uncovered a bug in Hales's graph enumeration code.

1 Introduction

In 1998, Hales announced the proof of the Kepler Conjecture (about the densest packing of congruent spheres in Euclidean space), which he published in a series of papers, ending with [6]. In addition to the sequence of journal articles, the proof employs three distinct large computations. To remove any doubt about the correctness of the proof due to the unverified computations, Hales started the Flyspeck project (http://code.google.com/p/flyspeck) to produce a formal proof of the Kepler Conjecture. An early contribution [17] was the verification of the enumeration of all potential counterexamples (i.e. denser packings) in the form of plane graphs, the so-called *tame* graphs. We confirmed that the so-called *archive* of tame graphs given by Hales is complete (in fact: too large). In a second step, it must be shown that none of these tame graphs constitute a counterexample. Obua [19] verified much of this part of the proof. This paper is about what happened when the geometry underlying the tame graph abstraction changed.

In 2009, Marchal [15] published a new and simpler approach to parts of Hales's proof. As a consequence Hales revised his proof. At the moment, only the online HOL Light theorems and proofs (most of Flyspeck is carried out in HOL Light [8]) reflect this revision (see the Flyspeck web page above). The complete revised proof will appear as a book [7]. Below we call the two versions of the proof the 1998 and the revised version.

The verified enumeration of tame graphs relies on executing functions verified in the theorem prover. Due to the revision, tame graph enumeration ran out of space because of the 10-fold increase in the number of tame graphs and the 100-fold increase in the overall number of graphs that need to be generated. Therefore I completely modularized that part of the proof in order to slot in more space-efficient enumeration machinery.

The verification of tame graph enumeration for the 1998 proof [17] was a bit of an anticlimax: at the end we could confirm that Hales's archive of tame graphs (which he had generated with an unverified Java program) was complete. Not so this time: I found two graphs that were missing from Hales's revised archive (which he had generated with a revised version of that Java program). A few days later Hales emailed me:

> *I found the bug in my code! It was in the code that uses symmetry to reduce the search space. This is a bug that goes all the way back to the 1998 proof. It is just a happy coincidence that there were no missed cases in the 1998 proof. This is a good example of the importance of formal proof in computer assisted proofs.*

The main contribution of this paper is an abstract description of the computational tools used in the revised proof, at a level where they can be reused in different settings. In particular, the paper provides the following generic components:

- A while combinator that allows to reason about terminating executions of partial functions without the need for a termination proof.
- Combinators for and beginnings of a theory of worklist algorithms.
- A stepwise implementation of an abstract type of sets over a quasi-order.
- A general schema for stepwise implementation of abstract data types in Isabelle's locale system of modules.
- A simple isomorphism checker for plane graphs.

Much of this paper is not concerned with the formal proof of the Kepler Conjecture per se, and those parts that are, complement [17].

2 Basics

This work is carried out with the Isabelle/HOL proof assistant. HOL conforms largely to everyday mathematical notation. This section summarizes nonstandard notation and features.

The function space is denoted by $\Rightarrow$. Type variables are denoted by $'a$, $'b$, etc. The notation $t :: \tau$ means that term t has type τ. Sets over type $'a$, type $'a\ set$ (which is just a synonym for $'a \Rightarrow bool$), follow the usual mathematical conventions. The image of a function over a set is denoted by $f\ `\ S$. Lists over type $'a$, type $'a\ list$, come with the empty list $[]$, the infix constructor $\cdot$, the infix @ that appends two lists, the length function $|.|$ and the conversion function *set* from lists to sets. Names ending in s typically refer to lists. The **datatype** $'a\ option = None \mid Some\ 'a$ is predefined. Implications are displayed either as arrows or as proof rules with horizontal lines.

Locales [2] are Isabelle's version of parameterized theories. A locale is a named context of functions $f_1, \ldots, f_n$ and assumptions $P_1, \ldots, P_m$ about them that is introduced roughly like this:

locale loc = **fixes** $f_1 \ldots f_n$ **assumes** $P_1 \ldots P_m$

The f_i's are the *parameters* of the locale. Every locale implicitly defines a predicate:

$$loc\ f_1 \ldots f_n \longleftrightarrow P_1 \wedge \ldots \wedge P_m$$

Locales can be hierarchical as in **locale** *loc'* = loc_1 + loc_2 + **fixes** ....

In the context of a locale, definitions can be made and theorems can be proved. This is called the *body* of the locale and can be extended dynamically.

An *interpretation* of a locale

interpretation $loc\ e_1\ \ldots\ e_n$

where the e_i's are expressions, generates the proof obligation $loc\ e_1\ \ldots\ e_n$ (recall that *loc* is a predicate), and, if the proof obligation is discharged by the user, one obtains all the definitions and theorems from the body of the locale, with each f_i replaced by e_i. For more details see the tutorial on locales [1].

Executability of functions in Isabelle/HOL does not rely on a constructive logic but merely on the ability of the user to phrase a function, by definition or by lemma, as a recursion equation [5]. Additionally we make heavy use of a preprocessor that replaces (by proof!) many kinds of bounded quantifications by suitable list combinators.

3 Worklist algorithms

3.1 While combinators

Proving termination of the enumeration of tame graphs is possible but tedious and unneccesary: after all, it does terminate eventually, which is proof enough. But how to define a potentially partial function in HOL? Originally I had solved that problem by brute force with *bounded recursion*: totalize the function with an additional argument of type *nat* that is decreased with every recursive call, return *None* if 0 is reached and *Some* r when the actual result r has been reached, and call the function with a sufficiently large initial number. It does the job but is inelegant because unnecessary. What are the alternatives?

Isabelle's standard function definition facility due to Krauss [10] does not require a termination proof, but until termination has been proved, the recursion equations are conditional and hence not executable. This is the opposite of the original while combinator defined in Isabelle/HOL [18]

$$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$$

where $'a$ is the "state" of the computation, the first parameter is the loop test, the second parameter the loop body that transforms the state, and the last parameter is the start state. This combinator obeys the recursion equation

$$while\ b\ c\ s = (\mathsf{if}\ b\ s\ \mathsf{then}\ while\ b\ c\ (c\ s)\ \mathsf{else}\ s)$$

and enables the definition of executable tail-recursive partial functions. But to prove anything useful about the result of the function, we first need to prove termination. This is a consequence of the type of *while*: the result does not tell us if it came out of a terminating computation sequence or is just some arbitrary value forced by the totality of the logic.

Krauss' recent work [11] theoretically provides what we need but the implementation does not yet. Hence Krauss and I defined a while combinator that returns an option value, telling us if it terminated or not, just as in bounded recursion outlined above, but without the need for a counter:

$$\textit{while-option} :: ('a \Rightarrow \textit{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ \textit{option}$$

$$\begin{array}{l}\textit{while-option}\ b\ c\ s = \\ (\textsf{if}\ \exists k.\ \neg\ b\ (c^k\ s)\ \textsf{then}\ \textit{Some}\ (c^{\textit{LEAST}\ k.\ \neg\ b\ (c^k\ s)}\ s)\ \textsf{else}\ \textit{None})\end{array}$$

It obeys a similar unfolding law as *while*

$$\textit{while-option}\ b\ c\ s = (\textsf{if}\ b\ s\ \textsf{then}\ \textit{while-option}\ b\ c\ (c\ s)\ \textsf{else}\ \textit{Some}\ s)$$

but allows to reason about *Some* results in the traditional manner of Hoare logic: invariants hold at the end if they hold at the beginning

$$\frac{\textit{while-option}\ b\ c\ s = \textit{Some}\ t \qquad \forall s.\ P\ s \wedge b\ s \longrightarrow P\ (c\ s) \qquad P\ s}{P\ t}$$

and at the end the loop condition no longer holds:

$$\textit{while-option}\ b\ c\ s = \textit{Some}\ t \Longrightarrow \neg\ b\ t$$

Note that termination is built into the premise *while-option* $b\ c\ s = \textit{Some}\ t$, which is the proposition that we intend to prove by evaluation.

Of course, if we can prove termination by deductive means, this ensures that *Some* result is returned:

$$\frac{\textit{wf}\ \{(t,\ s)\ |\ (P\ s \wedge b\ s) \wedge t = c\ s\} \qquad \forall s.\ P\ s \wedge b\ s \longrightarrow P\ (c\ s) \qquad P\ s}{\exists t.\ \textit{while-option}\ b\ c\ s = \textit{Some}\ t}$$

where *wf* R means that relation R is wellfounded, and where P is an invariant.

3.2 Worklist algorithms

Worklist algorithms repeatedly remove an item from the worklist, replace it by a new list of items, and process the item. They operate on pairs $(ws,\ s)$ of a worklist ws and a state s. We define

$$\begin{array}{l}\textit{worklist-aux}\ \textit{succs}\ f = \\ \textit{while-option}\ (\lambda(ws,\ s).\ ws \neq [])\\ \ (\lambda(ws,\ s).\ \textsf{case}\ ws\ \textsf{of}\ x{\cdot}ws' \Rightarrow (\textit{succs}\ s\ x\ @\ ws',\ f\ x\ s))\end{array}$$

of type
$('s \Rightarrow 'a \Rightarrow 'a\ list) \Rightarrow ('a \Rightarrow 's \Rightarrow 's) \Rightarrow 'a\ list \times 's \Rightarrow ('a\ list \times 's)\ option$.
Type $'a$ is the type of items, type $'s$ the type of states. Functions *succs* and f produce the next items and next state. If the algorithm terminates, it must have enumerated the set of items reachable via the successor function starting from the start items.

The successor function may depend on the state. This allows us, for example, to detect loops in the search process by carrying already visited items around in the state. But our application does not require this generality: its successor relationship is a tree. Hence we specialize *worklist-aux* and develop a theory of worklist algorithms on trees. A unified theory of worklist algorithms on trees and graphs is beyond the scope of this paper. From now on *succs* will not depend on the state and we define

$$worklist\text{-}tree\text{-}aux\ succs = worklist\text{-}aux\ (\lambda s.\ succs)$$

Upon termination the worklist will be empty and we project on the state:

$$\begin{array}{l} worklist\text{-}tree\ succs\ f\ ws\ s = \\ (\mathsf{case}\ worklist\text{-}tree\text{-}aux\ succs\ f\ (ws,\ s)\ \mathsf{of}\ None \Rightarrow None \\ \mid Some\ (ws,\ s) \Rightarrow Some\ s) \end{array}$$

In order to talk about the set of items reachable via *succs* we introduce the abbreviation

$$Rel\ succs \equiv \{(x,\ y) \mid y \in set\ (succs\ x)\}$$

that translates *succs* into a relation. In addition, $R\ ``\ S$ is the image of a relation over a set. Thus $(Rel\ succs)^*\ ``\ A$ is the set of items reachable from the set of items A via the reflexive transitive closure of *Rel succs*.

The first theorem about *worklist-tree* expresses that it folds f over the reachable items in some order:

$$\frac{worklist\text{-}tree\ succs\ f\ ws\ s = Some\ s'}{\exists rs.\ set\ rs = (Rel\ succs)^*\ ``\ set\ ws \wedge s' = fold\ f\ rs\ s}$$

where $fold\ f\ []\ s = s$ and $fold\ f\ (x{\cdot}xs)\ s = fold\ f\ xs\ (f\ x\ s)$.

This theorem is intentionally weak: *rs* is some list whose elements form the set of reachable items, in any order and with any number of repetitions. The order should not matter, to allow us to replace the particular depth-first traversal strategy of *worklist-aux* by any other, for example appending the successor items at the right end of the worklist. Of course it means that f should also be insensitive to the order. Moreover, f should be insensitive to duplicates, i.e. it should be idempotent. Thus this theorem is specialized for applications where the state is effectively a set of items, which is exactly what we are aiming for in our application, the enumeration and collection of graphs.

If we want to prove some property of the result of *worklist-tree*, we can do so by obtaining *rs* and proving the property by induction over *rs*. But we can also replace the induction by a Hoare-style invariance rule:

$$\frac{\begin{array}{c}\mathit{worklist\text{-}tree}\ \mathit{succs}\ f\ \mathit{ws}\ s = \mathit{Some}\ s' \\ \forall s.\ R\ []\ s\ s \qquad \forall r\ x\ \mathit{ws}\ s.\ R\ \mathit{ws}\ (f\ x\ s)\ r \longrightarrow R\ (x{\cdot}\mathit{ws})\ s\ r\end{array}}{\exists rs.\ \mathit{set}\ rs = (\mathit{Rel}\ \mathit{succs})^*\ ``\ \mathit{set}\ \mathit{ws} \wedge R\ rs\ s\ s'}$$

This rule is phrased in terms of a predicate R of type $'a\ list \Rightarrow 's \Rightarrow 's \Rightarrow bool$, where $R\ rs\ s\ s'$ should express the relationship between some start configuration $(rs,\ s)$ and the corresponding final state s', when the worklist has been emptied.

Unfortunately, this rule is too weak in practice: both the items and the states come with nontrivial invariants of their own, and the invariance of R can only be shown if we may assume that the item and state invariants hold for the arguments of R. In nice set theoretic language the extended rule looks like this:

$$\frac{\begin{array}{c}\mathit{worklist\text{-}tree}\ \mathit{succs}\ f\ \mathit{ws}\ s = \mathit{Some}\ s' \qquad \mathit{succs} \in I \to \mathit{lists}\ I \\ \mathit{set}\ \mathit{ws} \subseteq I \qquad s \in S \qquad f \in I \to S \to S \qquad \forall s.\ R\ []\ s\ s \\ \forall r\ x\ \mathit{ws}\ s.\ x \in I \wedge \mathit{set}\ \mathit{ws} \subseteq I \wedge s \in S \wedge R\ \mathit{ws}\ (f\ x\ s)\ r \longrightarrow R\ (x{\cdot}\mathit{ws})\ s\ r\end{array}}{\exists rs.\ \mathit{set}\ rs = (\mathit{Rel}\ \mathit{succs})^*\ ``\ \mathit{set}\ \mathit{ws} \wedge R\ rs\ s\ s'}$$

Here I and S are the invariants on items and states, *lists* I is the set of lists over I, and $A \to B$ is the set of functions from set A to set B.

As a simple application we obtain almost automatically that the function

$$\mathit{colls}\ \mathit{succs}\ P\ \mathit{ws} = \mathit{worklist\text{-}tree}\ \mathit{succs}\ (\lambda x\ xs.\ \mathit{if}\ P\ x\ \mathit{then}\ x{\cdot}xs\ \mathit{else}\ xs)\ \mathit{ws}\ []$$

indeed collects all reachable items that satisfy P:

$$\frac{\mathit{colls}\ \mathit{succs}\ P\ \mathit{ws} = \mathit{Some}\ \mathit{rs}}{\mathit{set}\ \mathit{rs} = \{x \in (\mathit{Rel}\ \mathit{succs})^*\ ``\ \mathit{set}\ \mathit{ws} \mid P\ x\}}$$

3.3 Sets over a quasi-order

In our application we need to collect a large set of graphs and we encounter many isomorphic copies of each graph. Storing all copies is out of the question for reasons of space. Hence we work with sets over a quasi-order, thus generalizing the graph isomorphism to a subsumption relation. We formulate our worklist algorithms in the context of an abstract interface to such a set data type and use Isabelle's locale mechanism (see §2) for this purpose. We start with a locale for the quasi-order, later to be interpreted by graph isomorphism:

locale *quasi-order* =
fixes *qle* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\preceq$ 60)
assumes $x \preceq x$
and $x \preceq y \Longrightarrow y \preceq z \Longrightarrow x \preceq z$

The following definitions are made in this context:

$x \in_\preceq M \equiv \exists y{\in}M.\ x \preceq y$
$M \subseteq_\preceq N \equiv \forall x{\in}M.\ x \in_\preceq N$
$M =_\preceq N \equiv M \subseteq_\preceq N \wedge N \subseteq_\preceq M$

The actual work will be done in the context of sets over $\preceq$ in locale *set-modulo*, an extension of locale *quasi-order*:

locale *set-modulo* = *quasi-order* +
fixes *empty* :: $'s$
and *insert-mod* :: $'a \Rightarrow 's \Rightarrow 's$
and *set-of* :: $'s \Rightarrow 'a$ *set*
and I :: $'a$ *set*
and S :: $'s$ *set*
assumes *set-of empty* = $\emptyset$
and $x \in I \Longrightarrow s \in S \Longrightarrow$ *set-of* $s \subseteq I \Longrightarrow$
set-of (*insert-mod* x s) = $\{x\} \cup$ *set-of* $s \vee$
$(\exists\, y{\in}$*set-of* $s.\ x \preceq y) \wedge$ *set-of* (*insert-mod* x s) = *set-of* s
and *empty* $\in S$
and $s \in S \Longrightarrow$ *insert-mod* x $s \in S$

In the body of a locale, the type variables in the types of the locale parameters are fixed. Above, $'a$ stands for the fixed element type, $'s$ for the abstract type of sets. The empty set is *empty*, elements are inserted by *insert-mod*. The sets I and S are invariants on elements and sets. In our application later on, both are needed. The behaviour of our sets is specified with the help of an abstraction function *set-of* that maps them back to HOL's mathematical sets.

The first assumption is clear, the last two assumptions state that all sets generated by *empty* and *insert-mod* satisfy the invariant. Only the second assumption needs an explanation, ignoring its self-explanatory preconditions. The point of this assumption is to leave the implementation complete freedom what to do when inserting a new element x. If the set already contains an element y that subsumes x, *insert-mod* is allowed to ignore x. But is not forced to: it may always insert x. This specification allows an implementation to choose how much effort to invest to avoid subsumed elements in a set. Because this subsumption test can be costly: in our application it involves testing for isomorphism with tens of thousands of graphs. Our implementation later on will use a hash function to zoom in on a small subset of potentially isomorphic graphs and only test isomorphism on those. This liberal specification of *insert-mod* saves us from proving that isomorphic graphs have the same hash value.

The above sets only offer *empty* and *insert-mod*, which seems overly toy-like or even useless. In reality there is also a function *all* :: $('a \Rightarrow bool) \Rightarrow 's \Rightarrow bool$ for examining sets, and many other functions could be added to *set-modulo*, but this would not raise any interesting new issues.

3.4 Collecting modulo subsumption

This subsection takes place completely within the context of locale *set-modulo*. We specialize the generic worklist combinator to fold *insert-mod* over the list of reachable items. At the same time we parameterize things further: we merely collect those items that satisfy some predicate P, and we don't collect the items themselves but apply some function f to them first:

$$\mathit{insert\text{-}mod2}\ P\ f\ x\ s = (\mathit{if}\ P\ x\ \mathit{then}\ \mathit{insert\text{-}mod}\ (f\ x)\ s\ \mathit{else}\ s)$$

Filtering with P in a separate pass is out of the question in our application because less than one 10^4th of all reachable items satisfy P; trying to store all reachable items would exhaust available memory. Applying f right away, rather than in a second pass, is also done for efficiency but is less critical.

The actual collecting is done by our worklist combinators:

$$\mathit{worklist\text{-}tree\text{-}coll\text{-}aux}\ \mathit{succs}\ P\ f = \mathit{worklist\text{-}tree}\ \mathit{succs}\ (\mathit{insert\text{-}mod2}\ P\ f)$$
$$\mathit{worklist\text{-}tree\text{-}coll}\ \mathit{succs}\ P\ f\ \mathit{ws} = \mathit{worklist\text{-}tree\text{-}coll\text{-}aux}\ \mathit{succs}\ P\ f\ \mathit{ws}\ \mathit{empty}$$

With the help of the generic theorems about *worklist-tree* (see §3.2) and the assumptions of locale *set-modulo* we can derive two important theorems about *worklist-tree-coll*. Its result is equivalent (modulo $\preceq$) to the image under f of those reachable items that satisfy P:

$$\frac{\mathit{worklist\text{-}tree\text{-}coll}\ \mathit{succs}\ P\ f\ \mathit{ws} = \mathit{Some}\ s' \qquad \mathit{succs} \in I_0 \to \mathit{lists}\ I_0 \qquad \mathit{set}\ \mathit{ws} \subseteq I_0 \qquad f \in I_0 \to I}{\mathit{set\text{-}of}\ s' =_{\preceq} f\ `\ \{x \in (\mathit{Rel}\ \mathit{succs})^*\ ``\ \mathit{set}\ \mathit{ws} \mid P\ x\}}$$

This theorem alone leaves the possibility that *set-of* s' contains elements that are only equivalent but not identical to the reachable items. But we can also derive

$$\frac{\mathit{worklist\text{-}tree\text{-}coll}\ \mathit{succs}\ P\ f\ \mathit{ws} = \mathit{Some}\ t \qquad \mathit{succs} \in I_0 \to \mathit{lists}\ I_0 \qquad \mathit{set}\ \mathit{ws} \subseteq I_0 \qquad f \in I_0 \to I}{\mathit{set\text{-}of}\ t \subseteq f\ `\ \{h \in (\mathit{Rel}\ \mathit{succs})^*\ ``\ \mathit{set}\ \mathit{ws} \mid P\ h\}}$$

This is helpful because it means, for example, that the resulting items all satisfy the invariant I.

4 Implementing sets modulo

We will now implement the interface *set-modulo* in two steps. First we implement *set-modulo* abstractly by another locale, *set-mod-maps*, with a map-like interface, and then we implement that by a concrete data structure, *Tries*. Figure 1 depicts the implementation relationships (where $A \to B$ means that A is implemented by B), and $\supseteq$ is locale extension. The full meaning of the diagram will become clear as we go along.

$$\begin{array}{ccccc} \mathit{set\text{-}modulo} & \to & \mathit{set\text{-}mod\text{-}maps} & \supseteq & \mathit{maps} \\ & & \downarrow & & \downarrow \\ & & \mathit{Graph} & & \mathit{Tries} \end{array}$$

Fig. 1. Implementation diagram

4.1 Maps

Our maps correspond to functions of type $'a \Rightarrow 'b\ list$ that return [] almost everywhere. We could implement them via ordinary maps of type $'a \Rightarrow 'c\ option$ as they are provided, for example, in the Collections Framework [12]. The latter did not exist yet when our proof was first developed, and since $'a \Rightarrow 'b\ list$ is simpler than $'a \Rightarrow 'b\ list\ option$ (where *None* acts like []) and of interest in its own right, this is what locale *maps* specifies:

locale *maps* =
fixes *empty* :: $'m$
and *up* :: $'m \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'm$
and *map-of* :: $'m \Rightarrow 'a \Rightarrow 'b\ list$
and *M* :: $'m\ set$
assumes *map-of empty* = $(\lambda a.\ [])$
and *map-of* (*up m a bs*) = *fun-upd* (*map-of m*) *a bs*
and *empty* $\in$ *M*
and $m \in M \Longrightarrow$ *up m a bs* $\in$ *M*

Type variable $'m$ represents the maps, $'a$ and $'b\ list$ its domain and range type. Maps are created from *empty* by *up* (update). Function *map-of* serves two purposes: as a lookup function and as an abstraction function from $'m$ to $'a \Rightarrow 'b\ list$. Function *fun-upd* is the predefined pointwise function update.

We extend *maps* with a function that produces the set of elements in the range of a map:

set-of m = $(\bigcup x\ set\ (map\text{-}of\ m\ x))$

4.2 Implementing sets modulo by maps

Before we present the details, we sketch the rough idea. Sets of elements of type $'b$ are represented by maps from $'a$ to $'b\ list$ where $'a$ is some type of "addresses" and there is some (hash) function *key* :: $'b \Rightarrow 'a$. Operation *insert-mod x m* will operate as follows: it looks up *key x* in *m*, obtains some list *ys*, checks if *x* is subsumed (modulo $\preceq$) by some element of *ys*, and adds it otherwise.

This abstract implementation is phrased as a locale that enriches *maps* with *key* and some further functions, and that will implement the operations of *set-modulo* with the help of those of *maps* and the newly added functions.

locale *set-mod-maps* = *maps* + *quasi-order* +
fixes *subsumed* :: $'b \Rightarrow 'b \Rightarrow bool$
and *key* :: $'b \Rightarrow 'a$
and *I* :: $'b\ set$
assumes $x{\in}I \Longrightarrow y{\in}I \Longrightarrow$ *subsumed x y* $\longleftrightarrow$ $(x \preceq y)$

The two parameters of *set-mod-maps* in addition to *key* are a predicate *subsumed*, meant to represent an executable version of the mathematical $\preceq$, and an invariant *I* that guarantees that *subsumed* coincides with $\preceq$ (see the assumption).

In the body of *set-mod-maps* the actual implementation of *insert-mod* is defined:

```
insert-mod x m =
(let k = key x; ys = map-of m k
 in if ∃ y∈set ys. subsumed x y then m else up m k (x·ys))
```

Now it is time to assert and prove that *set-mod-maps* is an implementation of *sets-modulo*, i.e. that we can interpret *sets-modulo* in the context of *set-mod-maps*:[1]

```
interpretation (in set-mod-maps)
  set-modulo (op ≼) empty insert-mod set-of I M
```

Predicate *set-modulo* takes six arguments. The first one is the quasi-order from locale *quasi-order* that it extends. Since *set-mod-maps* also extends *quasi-order*, we can supply that same relation $\preceq$. The other five parameters are the ones fixed in *set-modulo*: the empty set is interpreted by the empty map, *insert-mod* is interpreted by the *insert-mod* defined just above, *set-of* is interpreted by the *set-of* defined in the context of *maps*, the invariant on set elements is interpreted by the parameter *I*, and the set invariant by the maps invariant. The proofs of the *set-modulo* assumptions under this interpretation are straightforward.

Thus we have realized the horizontal arrow in Figure 1: any implementation (i.e. interpretation) of *set-mod-maps* yields an implementation of *set-modulo*.

4.3 Implementing maps by tries

Tries (pronounced as in "retrieval") are search trees indexed by lists of keys, one element for each level of the tree. Ordinary tries are found in the Collections Framework [12]; we provide a simple variation aimed at *maps*: lists rather than single items are stored, obviating the need for options.

Tries are defined in theory *Tries* and we refer to many of its operations with their qualified name, i.e. *Tries.f* rather than just *f*. The datatype itself is defined like this, where $'a$ is the type of keys and $'v$ the type of values:

```
datatype ('a, 'v) tries = Tries ('v list) (('a × ('a, 'v) tries) list)
```

The two projection functions are *values* (*Tries vs al*) = *vs* and *alist* (*Tries vs al*) = *al*. The name *alist* reflects that the second argument is an association list of keys and subtries. The invariant *Tries.inv* asserts that there are no two distinct elements with the same key in any alist in some trie. For concreteness, here is the code for *lookup* and *update*:

```
Tries.lookup t [] = values t
Tries.lookup t (a·as) =
(case map-of (alist t) a of None ⇒ [] | Some at ⇒ Tries.lookup at as)
```

[1] The actual syntax uses the keyword **sublocale** but we have chosen this more intuitive variation of **interpretation**.

Tries.update t [] *vs* = *Tries vs* (*alist t*)
Tries.update t (*a*·*as*) *vs* =
(*let tt* = *case map-of* (*alist t*) *a of None* ⇒ *Tries* [] [] | *Some at* ⇒ *at*
in Tries (*values t*) ((*a*, *Tries.update tt as vs*)·*rem-alist a* (*alist t*)))

Auxiliary functions are omitted and easy to reconstruct. Now it is straightforward to show

interpretation *maps* (*Tries* [] []) *Tries.update Tries.lookup Tries.inv*

Thus we have realized the arrow from *maps* to *Tries* in Figure 1. Once we also implement the *set-mod-map* extension (in §5.2), we obtain the body of *set-modulo*, in particular the collecting worklist algorithms and their correctness theorems.

4.4 Stepwise implementation in general

The above developments are instances of the following general schema, simplified for the sake of presentation. We want to implement an abstract interface

locale A = **fixes** f **assumes** P

In our case A is *set-modulo*. We base the implementation on n interfaces

locale B_i = **fixes** g_i **assumes** Q_i $\qquad\qquad (i = 1, \ldots, n)$

In our case, there are two B_i's: *maps*, and the extension of *set-mod-maps* with *key*, *subsumed* and *I*, which can be viewed as a separate locale that is added to the import list of *set-mod-maps*. Now, given some schema $F[g_1, \ldots, g_n]$ for defining f from the $g_1, \ldots, g_n$, A can be implemented by the B_i's:

locale *A-by-Bs* = B_1 + ... + B_n + **definition** *fimpl* = $F[g_1, \ldots, g_n]$

The correctness of this development step corresponds to the claim that A can be interpreted in the context of *A-by-Bs*, which of course needs to be proved:

interpretation (**in** *A-by-Bs*) *A fimpl*

Each B_i can either be implemented in the same manner as A, or it can be implemented directly:

interpretation B_i *concrete-*g_i

where *concrete-*g_i in an implementation of g_i on a suitable concrete type. In the end, we obtain an overall concrete implementation of A, together with an instance of the body of A.

It seems that this is the first time a general development scheme for abstract data types has been formulated for locales. The general idea of stepwise development of abstract data types via theory interpretations goes back to Maibaum *et al.* [14]. Theory interpretations are also a central concept in IMPS [3], but

with a focus on mathematics. Likewise, locales have primarily been motivated as a device for structuring mathematics [9]. Instances of the above schema can be found in a few large Isabelle developments, for example Lochbihler's Java-like language with threads [13], but even the Collections Framework by Lammich and Lochbihler [12], which is all about abstract data type specification and implementation, does not discuss the general picture and does not contain an instance of *A-by-Bs* above.

Similar specifications and developments are possible with the Coq module system, e.g. [4]. It would be interesting to investigate the precise relationship between locales and the Coq module system.

5 Application to plane graphs

As explained in the Introduction, Hales's proof involves the enumeration of a very large set of tame graphs, where tame graphs are by definition also plane [17]. Our representation of plane graphs follows Hales's 1998 proof: a plane graph is a set/list of faces, where each face is a list of vertices of type $'a$:

$'a\ Fgraph = 'a\ list\ set$
$'a\ fgraph = 'a\ list\ list$

Type *Fgraph* involves sets and belongs to the mathematical level, type *fgraph* represents sets by lists and belongs to the executable level. Below we develop a number of notions first on the *Fgraph* level and transfer them easily and directly to the *fgraph* level. We call graphs on both levels *face graphs* and frequently use the letter F for faces.

5.1 Plane graph isomorphisms

This subsection describes in some detail the isomorphism test that had to be left out of [17].

Face graphs need to be compared modulo rotation of faces and we define

$F_1 \cong F_2 \equiv \exists\, n.\ F_2 = rotate\ n\ F_1$
$\{\cong\} \equiv \{(F_1,\ F_2) \mid F_1 \cong F_2\}$

Relation $\{\cong\}$ is an equivalence and we can form the quotient $S\ //\ \{\cong\}$ with the predefined quotient operator $//$, defining proper homomorphisms and isomorphisms on face graphs, where φ is a function on vertices:

$is\text{-}pr\text{-}Hom\ \varphi\ Fs_1\ Fs_2 \equiv map\ \varphi\ `\ Fs_1\ //\ \{\cong\} = Fs_2\ //\ \{\cong\}$
$is\text{-}pr\text{-}Iso\ \varphi\ Fs_1\ Fs_2 \equiv is\text{-}pr\text{-}Hom\ \varphi\ Fs_1\ Fs_2 \wedge inj\text{-}on\ \varphi\ (\bigcup_{F \in Fs_1} set\ F)$
$is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2 \equiv is\text{-}pr\text{-}Iso\ \varphi\ (set\ Fs_1)\ (set\ Fs_2)$

The first two functions operate on type *Fgraph*, the last one on *fgraph*. The attribute "proper" indicates that orientation of faces matters. The more liberal version where the faces of one graph may have the reverse orientation of those of the other graph, which corresponds to the standard notion of graph isomorphism, is easily defined on top:

$$\textit{is-Iso}\ \varphi\ Fs_1\ Fs_2 \equiv \textit{is-pr-Iso}\ \varphi\ Fs_1\ Fs_2 \vee \textit{is-pr-Iso}\ \varphi\ Fs_1\ (\textit{rev}\ `\ Fs_2)$$
$$\textit{is-iso}\ \varphi\ Fs_1\ Fs_2 \equiv \textit{is-Iso}\ \varphi\ (\textit{set}\ Fs_1)\ (\textit{set}\ Fs_2)$$
$$g_1 \simeq g_2 \equiv \exists\, \varphi.\ \textit{is-iso}\ \varphi\ g_1\ g_2$$

What we need is an executable isomorphism test. A simple solution would have been to search for an isomorphism by some unverified function and check the result with a verified checker. Although this is a perfectly reasonable solution, we wanted to see if a verified isomorphism test that performs well in our context is also within easy reach. It turns out it is. The verification took of the order of 600 lines of proof that rely heavily on automation of set theory.

We start with the search for a proper isomorphism. The isomorphism is represented by a list of vertex pairs I that is built up incrementally. Given two lists of faces, repeatedly take a face F_1 from the first list, find a matching face F_2 in the second list, and remove both faces. Matching means that F_1 and F_2 must have the same length, and for some $n < |F_2|$, the bijection obtained by pairing off F_1 and *rotate* n F_2 vertex by vertex is compatible with I. Then we can merge it with I. This is the corresponding recursive function:

$$\textit{pr-iso-test-rec} :: ('a \times 'b)\textit{list} \Rightarrow 'a\ \textit{fgraph} \Rightarrow 'b\ \textit{fgraph} \Rightarrow \textit{bool}$$

$$\begin{array}{l}
\textit{pr-iso-test-rec}\ I\ []\ Fs_2 \longleftrightarrow Fs_2 = [] \\
\textit{pr-iso-test-rec}\ I\ (F_1{\cdot}Fs_1)\ Fs_2 \longleftrightarrow \\
(\exists\, F_2{\in}\textit{set}\ Fs_2. \\
\quad |F_1| = |F_2| \wedge \\
\quad (\exists\, n{<}|F_2|. \\
\qquad \textsf{let}\ I' = \textit{zip}\ F_1\ (\textit{rotate}\ n\ F_2) \\
\qquad \textsf{in}\ \textit{compat}\ I'\ I \wedge \\
\qquad\quad \textit{pr-iso-test-rec}\ (\textit{merge}\ I'\ I)\ Fs_1\ (\textit{remove1}\ F_2\ Fs_2)))
\end{array}$$

Function *compat* checks if two lists of vertex pairs are compatible

$$\textit{compat}\ I\ I' \equiv \forall\, (x,\ y){\in}\textit{set}\ I.\ \forall\, (x',\ y'){\in}\textit{set}\ I'.\ x = x' \longleftrightarrow y = y'$$

and function *merge* merges them:

$$\begin{array}{l}
\textit{merge}\ []\ I = I \\
\textit{merge}\ (xy{\cdot}xys)\ I = \\
(\textsf{let}\ (x,\ y) = xy \\
\ \textsf{in if}\ \forall\, (x',\ y'){\in}\textit{set}\ I.\ x \neq x'\ \textsf{then}\ xy{\cdot}\textit{merge}\ xys\ I\ \textsf{else}\ \textit{merge}\ xys\ I)
\end{array}$$

Moving from proper isomorphism to isomorphism is easy

$$\textit{iso-test}\ g_1\ g_2 \longleftrightarrow \textit{pr-iso-test}\ g_1\ g_2 \vee \textit{pr-iso-test}\ g_1\ (\textit{map}\ \textit{rev}\ g_2)$$

where $\textit{pr-iso-test}\ Fs_1\ Fs_2 \longleftrightarrow \textit{pr-iso-test-rec}\ []\ Fs_1\ Fs_2$.

Function *pr-iso-test-rec* is the result of a stepwise development that we skip in favour of the final correctness theorem:

$$\textit{pr-iso-test-rec}\ []\ Fs_1\ Fs_2 \longleftrightarrow (\exists\, \varphi.\ \textit{is-pr-iso}\ \varphi\ Fs_1\ Fs_2)$$

This theorem comes with a number of preconditions on the two face lists: in each face, all vertices must be distinct, all faces in each list must be distinct modulo $\cong$, and the empty face is not allowed. An executable version of these preconditions, for the case where the vertices are natural numbers, can be expressed like this:

$$\textit{pre-iso-test Fs} \longleftrightarrow$$
$$[] \notin \textit{set Fs} \wedge (\forall F{\in}\textit{set Fs}.\ \textit{distinct } F) \wedge \textit{distinct } (\textit{map rotate-min Fs})$$

Function *rotate-min* produces a unique representative of the $\cong$ equivalence class of a face by rotating the minimal vertex to the head of the list.

The key theorem now states that under the executable preconditions, the executable and the mathematical definition of isomorphism agree:

$$\textit{pre-iso-test } Fs_1 \Longrightarrow \textit{pre-iso-test } Fs_2 \Longrightarrow \textit{iso-test } Fs_1\ Fs_2 \longleftrightarrow Fs_1 \simeq Fs_2$$

5.2 Sets of graphs modulo isomorphism

Now we can reap the benefits of the implementation work in §3.4. The interpretation of locale *quasi-order* with $\simeq$ on type *fgraph* is trivial and omitted. The interpretation of *set-mod-maps* is more involved:

interpretation
set-mod-maps (*Tries* [] []) *Tries.update Tries.lookup Tries.inv*
(*op* $\simeq$) *iso-test hash pre-iso-test*

The first four parameters are identical to those in the interpretation of the sublocale *maps* in §4.2. The quasi-order is interpreted as $\simeq$. The last three parameters interpret the *subsumed*, *key* and I parameters of *set-mod-maps*. Only the hash function remains to be explained, informally. It takes an *fgraph* and produces a list of natural numbers, in this order: the number of vertices, the number of faces, the sorted list of degrees of each vertex. All of these are well-known invariants under isomorphism, but as explained in §3.3, we have set things up such that we do not need to prove this.

The final theorem in the previous subsection is all we need to prove the one assumption of locale *set-mod-maps* in this interpretation. Now we have established the last remaining arrow in Figure 1, the one from to *set-mod-maps* to *Graph* (representing the plane graph theory).

5.3 Application to Hales's proof

During the enumeration, graphs are represented by an abstract type *graph* with a successor function *next* :: *graph* $\Rightarrow$ *graph list*, a predicate *final* that picks out the tame graphs, and a projection *fgraph* :: *graph* $\Rightarrow$ *nat fgraph*.

The interpretation of *set-mod-maps* above yields a function *worklist-tree-coll* that we can specialize as follows to the enumeration of all tame graphs:

worklist-tree-coll next final fgraph

of type *graph list* ⇒ (*nat*,*nat fgraph*) *tries option*. In the end, this function is applied to four different start graphs, runs for 11 hours, and terminates with *Some* tries; the resulting tries are compared modulo isomorphism with an archive of tame graphs, which Hales had initially supplied. The first time the verified enumeration terminated successfully, I found that the archive lacked two graphs. The completed archive is available online, as are all the Isabelle theories [16].

During the enumeration, a total of 1 870 507 512 graphs are generated, of which 348 231 are final tame graphs, of which 18 762 are distinct modulo isomorphism. The final tame graphs have at most 25 faces (18.6 on average) and at most 15 vertices (13.8 on average). Our hash function works very well: on average, there are 3.1 graphs in each entry of a trie, and in the worst case there are 97.

6 Conclusion

This work is an encouraging example of both the contribution that theorem proving can make to extreme mathematical proofs and the contribution that software development methods can make to theorem proving. Initially we had hacked our way through the proof and did not describe the details in [17]. This paper is a rational reconstruction of the underlying data structures and algorithms of that hack. This exercise in modularization has given rise to a number of interesting reusable components.

Acknowledgement I would like to thank Tom Hales for hosting my visit to Pitt, Jasmin Blanchette for grammatical and stylistic scrutiny, and Alex Krauss for the subsumption relation and many other improvements.

References

1. Clemens Ballarin. Tutorial to locales and locale interpretation. http://isabelle.in.tum.de/dist/Isabelle2011/doc/locales.pdf.
2. Clemens Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. Borwein and W. Farmer, editors, *Mathematical Knowledge Management (MKM 2006)*, volume 4108 of *Lect. Notes in Comp. Sci.*, pages 31–43. Springer-Verlag, 2006.
3. William Farmer. Theory interpretation in simple type theory. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic and Term Rewriting. (HOA '93)*, volume 816 of *Lect. Notes in Comp. Sci.*, pages 96–123. Springer-Verlag, 1994.
4. Jean-Christophe Filliâtre and Pierre Letouzey. Functors for proofs and programs. In David A. Schmidt, editor, *European Symposium on Programming (ESOP 2004)*, volume 2986 of *Lect. Notes in Comp. Sci.*, pages 370–384. Springer-Verlag, 2004.
5. Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *Lect. Notes in Comp. Sci.*, pages 103–117. Springer-Verlag, 2010.

6. Thomas C. Hales. Sphere packings, VI. Tame graphs and linear programs. *Discrete and Computational Geometry*, 36:205–265, 2006.
7. Thomas C. Hales. *Dense sphere packings: A blueprint for formal proofs*. Cambridge University Press, forthcoming.
8. John Harrison. HOL Light: An overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lect. Notes in Comp. Sci.*, pages 60–66. Springer-Verlag, 2009.
9. Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales: A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics, TPHOLs'99*, volume 1690 of *Lect. Notes in Comp. Sci.*, pages 149–165. Springer-Verlag, 1999.
10. Alexander Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Automated Reasoning*, 44:303–336, 2010.
11. Alexander Krauss. Recursive definitions of monadic functions. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *Workshop on Partiality and Recursion in Interactive Theorem Proving (PAR 2010)*, volume 43 of *EPTCS*, pages 1–13, 2010.
12. Peter Lammich and Andreas Lochbihler. The Isabelle Collections Framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *Lect. Notes in Comp. Sci.*, pages 339–354. Springer-Verlag, 2010.
13. Andreas Lochbihler. Jinja with threads. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sourceforge.net/entries/JinjaThreads.shtml, 2007. Formal proof development.
14. T. S. E. Maibaum, Paulo A. S. Veloso, and M. R. Sadler. A theory of abstract data types for program development: Bridging the gap? In *TAPSOFT, Volume 2*, volume 186 of *Lect. Notes in Comp. Sci.*, pages 214–230. Springer-Verlag, 1985.
15. Christian Marchal. Study of the Kepler's conjecture: the problem of the closest packing. *Mathematische Zeitschrift*, Published online 2009.
16. Tobias Nipkow. Flyspeck I: Tame graphs. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/devel-entries/Flyspeck-Tame.shtml, 2011. Formal proof development.
17. Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lect. Notes in Comp. Sci.*, pages 21–35. Springer-Verlag, 2006.
18. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002.
19. Steven Obua and Tobias Nipkow. Flyspeck II: The basic linear programs. *Annals of Mathematics and Artificial Intelligence*, 56:245–272, 2009.