Ordinal Arithmetic: A Case Study for Rippling in a Higher Order Domain

Louise A. Dennis and Alan Smaill

Division of Informatics, University of Edinburgh, 80 South Bridge, Edinburgh, EH1 1HN, UK, louised@dai.ed.ac.uk, A.Smaill@ed.ac.uk

Abstract. This paper reports a case study in the use of proof planning in the context of higher order syntax. Rippling is a heuristic for guiding rewriting steps in induction that has been used successfully in proof planning inductive proofs using first order representations. Ordinal arithmetic provides a natural set of higher order examples on which transfinite induction may be attempted using rippling. Previously Boyer-Moore style automation could not be applied to such domains. We demonstrate that a higher-order extension of the rippling heuristic is sufficient to plan such proofs automatically. Accordingly, ordinal arithmetic has been implemented in λ*Clam*, a higher order proof planning system for induction, and standard undergraduate text book problems have been successfully planned. We show the synthesis of a fixpoint for normal ordinal functions which demonstrates how our automation could be extended to produce more interesting results than the textbook examples tried so far.

1 Introduction

This paper reports on using λ*Clam* to plan proofs about ordinal arithmetic, making use of higher order features. Ordinal arithmetic is of interest, among other things, for its use in aiding termination proofs (see [7]), and in classifying proof-theoretic complexity.

The system was used to attempt to plan proofs of examples appearing in a number of text books [21, 9, 13] with encouraging results. We were also able to synthesize a fixpoint for a normal function. The emphasis is on the automated control of proof search, and we aim for both a declarative account of this control, and the construction of mathematically natural proofs. As far as we know this is the first time that automation of such proofs has been attempted in a fully higher-order fashion.

It was found that no modifications of the existing proof plan for induction and theory of rippling were needed to plan these higher order proofs. This result supports the generality of the rippling approach, and its ability to generate mathematically natural proofs.

This paper begins with an overview of ordinal arithmetic (§2) and proof planning focusing on the proof plan for induction using the ripple heuristic as implemented in λ*Clam* (§3). We then discuss the modifications we had to make to this in order to handle ordinal arithmetic (§4) and evaluate our results (§5). Finally we consider some options for further work (§6).

2 Ordinal Arithmetic: A Higher Order Family of Problems

First we present the background to the problem area. Traditionally the theory of ordinals is presented within set theory; a machine-checked presentation following this route is given in [17]. We instead present a computationally more direct theory of ordinals in which to reason.

Ordinals can be thought of as (equivalence classes of) well-ordered linear orders. As such they are a generalisation of the usual natural numbers. We work with so-called *ordinal notations*, as in [14]; this formalisation was carried out in the Lego theorem prover [15] working in a constructive type theory. These notations give us a language that allows us to speak about some (but not all) of the ordinals. An inductively defined type for ordinal notations is given by Coquand in [8] as follows:

```
datatype ordinal = 0 | s of ordinal | sup of (nat -> ordinal)
```

where we understand the `Sup` operation to refer to the least upper bound of the ordinals in the range of its argument. This definition yields corresponding induction and recursion schemes as follows.

Theorem 1 (Induction for ordinal notations). *Suppose that*

1. $\phi(0)$;
2. *for every* α, *if* $\phi(\alpha)$ *then* $\phi(s(\alpha))$;
3. *for every function* $f : nat \to ordinal$, *if for every* $n : nat$, $\phi(n)$ *then* $\phi(sup(f))$.

Then for every ordinal notation α, $\phi(\alpha)$.

This makes ordinals a credible test case for inductive theorem provers. Proofs using this principle are naturally higher-order, as we will see.

A corresponding iteration principle is then available to us, allowing us to define operations involving ordinal notations. Suppose that for some type T we have $a : T$, $G : T \to T$ and $H : (nat \to T) \to T$. Then the iteration principle yields a function $F : ordinal \to T$ such that:

$$F(0) = a, \tag{1}$$

$$F(s(x)) = G(F(x)), \tag{2}$$

$$F(sup(f)) = H(f). \tag{3}$$

Note that equality here is between ordinal notations – this is a finer equality than that between ordinals where isomorphic orders are identified. However, it is sufficient for present purposes.

To take a particular example, addition for ordinals is defined as follows:

$$x + 0 = x, \tag{4}$$

$$x + s(y) = s(x + y), \tag{5}$$

$$x + sup(f) = sup(\lambda n.\, x + f(n)). \tag{6}$$

We can embed the natural numbers in the ordinals in the obvious way via a function *embed*; the ordinal ω is then $sup(embed)$.

Since for an infinite order, tacking an extra element at the beginning or at the end can give different results, in the sense of having different properties as orders, not all the arithmetic identities for natural numbers still hold for ordinals (commutativity of addition fails for example), and we have to be more careful about the arguments used for recursive definitions. Halmos [12] gives a good account of the ideas involved.

In brief, ordinal arithmetic presents itself as an area where induction applies but one which contains higher order aspects. Not all theorems in standard arithmetic hold for ordinal numbers making it genuinely distinct from the finite case.

3 Proof Planning

Proof planning was first suggested by Bundy [4]. A proof plan is a proof of a theorem, at some level of abstraction, presented as a tree. A proof plan is generated using AI-style planning techniques. The planning operators used by a proof planner are the *proof methods*. These are defined by their pre- and post-conditions which are used by the planner to form the proof plan. The defining feature of a proof method is this presentation in terms of pre- and post- conditions. Methods themselves need not contain any guarantee of soundness and generally refer to a tactic in an LCF-style theorem prover which can be run to create a sound proof. In principle a method could directly include a justification function etc. in which case it would be both a method and a tactic. There is, in our opinion, no reason why methods and tactics must necessarily be distinct. On the other hand proof methods are a general framework for coordinating proof steps and so are not specifically tied to the LCF paradigm.

The first and most famous proof plan is that for induction with the associated rippling heuristic (a form of rewriting constrained to be terminating by meta-logical annotations). This was implemented in the *Clam* proof planner. λ*Clam* [19] is a higher order descendant of *Clam* and was the chosen system for this case study.

3.1 Proof Planning in λ*Clam*

Proof planning in λ*Clam* works as follows: A goal is presented to the system. This goal is a sequent and several of the methods embody standard sequent calculus inference rules. There are two sorts of method; compound and atomic. A compound method is a *methodical expression* built from methods and *methodicals*. Methodicals are analogous to tacticals in an LCF setting. They specify that, for instance, one method should be applied then another, or a method should be repeated as many times as possible. Each compound method thus imposes a structure on its *submethods*. In this way the `step_case` method for induction attempts to ripple (rewrite) the induction conclusion at least once and does not attempt any other theorem proving method (thus reducing the search space at this point) and then tries to fertilise (exploit the induction hypothesis) once it is no longer possible to rewrite the conclusion. Atomic methods have the normal preconditions and effects (postconditions)[1] presentation. If all of an atomic method's preconditions are satisfied then it is applied and the effects are used to determine the new goal on which planning should be attempted.

λ*clam*[2] is implemented in λProlog, a higher order, strongly typed, modular logic programming language. λProlog is used to provide a declarative language for method precondition descriptions.

3.2 The Proof Strategy for Induction and Rippling

The proof strategy for induction can be seen in figure 1. This is implemented as a selection of atomic and compound methods. The diagram shows a top level repeat which attempts a disjunction of methods. These include basic tautology checking, generalisation of common subterms and also symbolic evaluation and the induction strategy (`ind_strat`). Within the induction strategy, the induction method performs ripple analysis [6] to choose an induction scheme (from a selection specified in λ*Clam*'s theories) and produces subgoals for base and step cases. The base cases are passed out to the repeat of the top level strategy. The step cases are annotated with *skeletons* and *embeddings* (described below) and then the wave method is repeatedly applied to them followed by fertilisation (exploitation of the induction hypothesis). Annotations are then removed. The methods (`set_up_ripple` and `post_ripple`) which place and remove annotations are omitted from the diagram. The results are then passed out to the top level strategy again. The process terminates when all subgoals have been reduced to *true* (or *false* in the case of failure).

We will discuss the `wave_method` in more detail since it was a new formulation of this method we wished to investigate. The wave method embodies the rippling heuristic. Rippling was first introduced in [5]. We use the theory as presented by Smaill & Green [20] who proposed a version that naturally coped with higher-order features. Rippling steps apply rewrite rules to a target term which is associated with a skeleton and an embedding that relates the skeleton to the target term (e.g. rippling rewrites an induction conclusion which has an induction hypothesis embedded in it). In the present context, we make use of higher order rewriting, in the style of [11]. After rewriting a new embedding of the skeleton into the rewritten term is calculated. There is a measure on embeddings and any rewriting step must reduce this *embedding measure* (written as $<_\mu$). This is a generalisation of the original version of rippling that used annotated *wave rules* to rewrite annotated terms.

[1] In λ*Clam* method postconditions are entirely concerned with generating the next subgoal (the output) – hence they are frequently called effects to illustrate their more restricted role.

[2] Details of λ*Clam* are at `http://dream.dai.ed.ac.uk/systems/lambda-clam/`. The current implementation can be obtained by emailing `dream@dai.ed.ac.uk`.

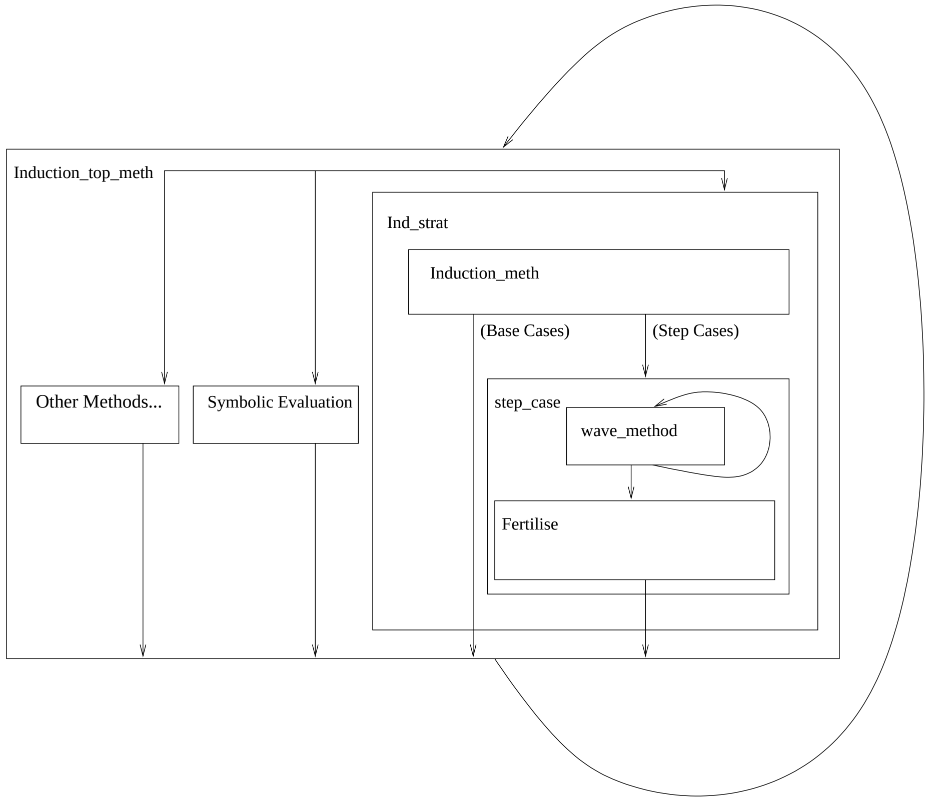

Fig. 1. The Proof Strategy for Induction

Rippling is terminating [2]. Rippling either moves differences outwards in the term structure so that they can be cancelled away or inwards so that the differences surround a universally quantified variable (or *sink*). If it is possible to move differences inwards in this way the embedding is said to be *sinkable*. The measure on embeddings allows differences that are being moved outwards to be moved inwards but not vice versa – this is at the heart of the guarantee of termination.

The `wave_method` method has five preconditions. It finds a rewrite rule that rewrites the goal. It then checks that there is still an embedding of the skeleton into the rewritten goal and that this new embedding is less, according to the embedding measure, than the original embedding. It checks that the embedding is sinkable and that any conditions for the application rule are trivial. This is shown in figure 2. The method will backtrack in order to try to satisfy all requirements, and if it is successful returns a new goal.

Input

$$ripple\_goal(H \vdash G, S, E).$$

where *ripple_goal* is a triple of a sequent, a skeleton, S and an embedding, E, of that skeleton into the goal, G.

Preconditions

1. The conditional rewrite rule *Rule*, $Cond \rightarrow X \Leftarrow:\Rightarrow Y$ instantiated with some substitution σ, applies to G and rewrites it to G'.
2. There is an embedding E' that embeds S in G'.
3. $E' <_{\mu} E$.
4. $\sigma(Cond) = True$ or $\sigma(Cond) \in H$.
5. E' is sinkable.

Output

$$ripple\_goal(H \vdash G', S, E')$$

Fig. 2. The Wave Method

Embeddings We review here the notion of embeddings from Smaill & Green [20]. These provide the higher-order framework for rippling used in this paper.

Embeddings are described by a tree data structure. Embedding trees describe how a *skeleton* embeds in a term, called the *erasure*. The nodes in an embedding tree can be viewed as labels on the nodes in the term tree of the skeleton. These labels contain addresses and directions. The directions are used during rippling as outlined above. The addresses are the addresses of nodes in the term tree of the erasure which correspond to that node in the skeleton. Term trees represent function application and λ-abstraction explicitly as nodes with constant and variables symbols appearing only at the leaves of the tree. Our implementation also contains tuple nodes for lists of arguments to functions but these are not necessary to the theory. Embeddings do not annotate λ-abstraction nodes. Where an embedding matches a variable in the skeleton to one in the erasure it indicates that they are α-convertible. It is the ability to coherently handle λ-abstractions which was particularly valuable in this experiment. The ability to handle difference occuring within functions as well as the arguments to functions is also an extension of the previous calculus.

Example 1. Consider embedding the term $\lambda x.\ f(x)$ into the term $\lambda y.\ \lambda x.\ (f(y)+x)$. We do this as in figure 3. The two terms are shown as trees with branches represented by solid lines. The address of each node is given (λ-abstraction nodes do not carry addresses). The embedding appears between them as an embedding tree with dashed lines – the address label of the nodes is also shown. The dotted arrows illustrate how the embedding tree links the two terms.

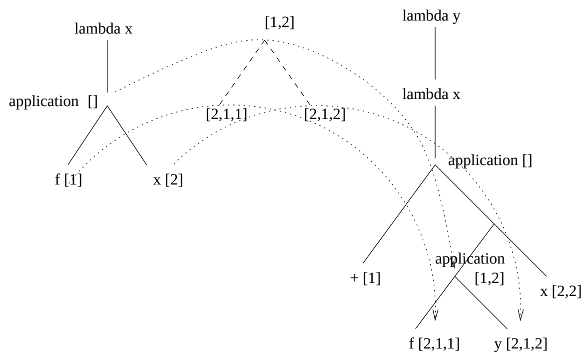

Fig. 3. An Embedding

The embedding tree for this is (node [1, 2] [(leaf [1, 1, 1]) (leaf [2, 1, 2])])). This states that the function application at the top of $f(x)$ matches with the node at address [1, 2] of $f(y) + x$ (i.e. the application involving $+$ has been bypassed), that the function symbol f matches the sub term at [2,1,1] (i.e. f) and x matches y (i.e. the bound variables can be consistently renamed in either (or both) terms so that they are identical).

The annotations originally used in rippling are still useful for presentation. Annotations consist of contexts (expressions with holes) indicated by a wave front (box) with a directional arrow. The holes in the context are wave holes (i.e. they are filled with an expression which is underlined). The skeleton is everything that appears outside wave fronts, or in wave holes. So the above embedding can be presented as

$$\lambda x.\ \boxed{\lambda y.\ \underline{f(y)} + x}^{\uparrow}.$$

NB. It is important to remember that the annotations do not actually exist in the implementation which records instead the skeleton and embedding[3]. In some cases it is less easy to use the traditional annotations to represent embeddings, particularly when dealing with λ-abstrations which are ignored by the embedding trees. You can see that in the above the bound variables in the skeleton are assumed to have been appropriately renamed already. These annotations are just a presentational convenience.

[3] In fact λ*Clam* maintains a list of possible embeddings during rippling since there may be more than one way to embed the induction hypotheses into the conclusion. For convenience we assume here there is only one.

4 Proof Planning Ordinal Arithmetic

To implement ordinal arithmetic in λ*Clam* we faced a number of tasks.

1. Provide a representation of ordinals, including limit ordinals.
2. Provide, as rewrite rules, definitions for the standard arithmetical operators.
3. Determine an appropriate small set of support lemmas and provide these as additional rewrite rules.
4. Provide one or more induction schemes for transfinite induction.

The notion of rewriting here is higher-order, in the sense of [1]. Thus variables in rewrite rules may be of functional type.

4.1 Representing Ordinals in λ*Clam*

It is simple to supply new constants and types for λ*Clam* in an appropriate module.

We know that 0 is an ordinal and that there are two ways of forming larger ordinals, either by applying the successor function, or taking the supremum (limit) of a set of ordinals. So we were able to use these as constructors for a type *ordl*.

We chose to use the notation lim (the "limit" of the function F) rather than *sup* in the implementation. In fact, the definitional rewrite rules for plus etc. express the continuity of the standard operations, with respect to the standard topology on ordinals; however, the reader should understand uses of lim in what follows to correspond to the traditional supremum or set-theoretic union.

4.2 Defining the Arithmetical Operators

The first two clauses in the definition of plus are similar to the standard definition for natural numbers except that the recursion is defined on a non-standard argument:

$$x + 0 = x, \tag{7}$$
$$x + s(y) = s(x + y), \tag{8}$$

These become the rewrite rules

$$X + 0 \Leftarrow:\Rightarrow X, \tag{9}$$
$$X + s(Y) \Leftarrow:\Rightarrow s(X + Y), \tag{10}$$

in λ*Clam*. These rules can be used in either direction during rippling but only from left to right in symbolic evaluation.

λ*Clam* allows a user to state exactly which lemmas and definitions they wish to use. So the definitions were simply presented to the system as axioms in a special module for ordinals. This does place a burden on the user, however, to use only a consistent set of rewrites. We ensured this with a Lego construction of our rules. Our proof methods are all consistent with constructive type theory so the resulting plans represent proofs in constructive logic.

The final defining equation for addition is as follows:

$$x + \lim f = \lim \lambda n.\ (x + f(n)).$$

This then yields the rewrite rule

$$X + \lim F \Leftarrow:\Rightarrow \lim \lambda y.\ (X + F(y)) \tag{11}$$

in λ*Clam*. Multiplication and exponentiation were given additional rewrite rules in the same way.

4.3 Defining Support Lemmas

We attempted to use as minimal a set of consistent support lemmas as possible. These include a handful of cancellation rules (e.g. $(s(X) = s(Y)) \Leftarrow:\Rightarrow (X = Y))$ and identity $((X = X) \Leftarrow:\Rightarrow \mathit{true})$. We also introduced the following new rewrites that correspond to basic properties of ordinal functions.

$$\lim F = \lim G :\Rightarrow (\forall n : \mathit{nat}.\ F(n) = G(n)), \tag{12}$$

These rewrites are derived from implications not equations and so may only be used in one direction during proof.

Finally, we allow the swapping of universal quantifiers; the rippling mechanism ensures that this will not produce looping in the proof search.

$$\forall x : T\ \forall y : U\ \Phi(x, y) \Leftarrow:\Rightarrow \forall y : U\ \forall x : T\ \Phi(x, y) \tag{13}$$

4.4 An Induction Scheme for Transfinite Induction

The induction scheme used follows from the definition of the data-type of ordinal notations and was developed in Lego. For reference this is:

$$\frac{P(0)\quad \forall x.\, P(x) \vdash P(s(x))\quad \forall f : \mathit{nat} \to \mathit{ordl}.\, (\forall n : \mathit{nat}.\, P(f(n))) \vdash P(\lim \lambda x.\, f(x))}{\forall \alpha : \mathit{ordl}.\, P(\alpha)} \tag{14}$$

This was added into the system in the ordinal module.

5 Evaluation

This machinery was enough to plan automatically the following goals. The times given are for a Sun E450 server dual processor machine with 2G of memory.

Theorem	Time Taken (CPU mins:secs)
$\forall \alpha : ordl.\, 0 + \alpha = \alpha,$	3.8
$\forall \alpha : ordl.\, \alpha + 1 = s(\alpha),$	0.8
$\forall \alpha, \beta, \gamma : ordl.\, (\alpha + \beta) + \gamma = \alpha + (\beta + \gamma),$	18.1
$\forall \alpha : ordl.\, 1 \cdot \alpha = \alpha,$	7.2
$\forall \alpha : ordl.\, \alpha \cdot 1 = \alpha,$	7.2
$\forall \alpha : ordl.\, \alpha \cdot 2 = \alpha + \alpha,$	5.0
$\forall \alpha, \beta, \gamma : ordl.\, \alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma,$	1:09.5
$\forall \alpha, \beta, \gamma : ordl.\, (\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma),$	2:20.9
$\forall \alpha : ordl.\, \alpha^1 = \alpha,$	8.3
$\forall \alpha : ordl.\, \alpha^2 = \alpha \cdot \alpha,$	8.7
$\forall \alpha, \beta, \gamma : ordl.\, \alpha^{\beta+\gamma} = \alpha^\beta \cdot \alpha^\gamma,$	4:49.0
$\forall \alpha, \beta, \gamma : ordl.\, \alpha^{\beta \cdot \gamma} = \alpha^{\beta^\gamma}.$	7:04.9

These were drawn from exercises in three undergraduate text books [21, 9, 13]. They represent examples of ordinal arithmetic listed in those books which use only transfinite induction and the definitions of the arithmetical operators (i.e. which don't also use some set theoretic reasoning based on the definitions of ordinals) except for those involving the order relation. They were planned using only the given induction scheme, definitions and lemmas mentioned in section 4.

5.1 The Distributivity of Multiplication over Addition

As a sample of the sort of plan produced by λ*Clam* for these examples we present the plan for the distributivity of multiplication over addition. It is presented in a natural language way, but each step performed by the proof planner is marked.

Example 2. **Goal:**

λ*Clam* performs backwards proof so it starts with the initial goal.

$$\forall \alpha, \beta, \gamma : ordl.\, \alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma. \tag{15}$$

`induction_top_meth` is applied to this. This tries each of its submethods in turn. Only one of these succeeds, `ind_strat`, because its submethod, `induction_meth` succeeds. Ripple analysis[4] suggests the use of induction on γ. This splits the goal into three subgoals; one base case and two step cases.

Case 1:

$$\forall \alpha, \beta : ordl.\, \alpha \cdot (\beta + 0) = \alpha \cdot \beta + \alpha \cdot 0. \tag{16}$$

This is a base case and so is returned to `induction_top_meth` unaltered. This time symbolic evaluation succeeds and performs the following sequence of

[4] Ripple analysis analyses possible rewrites and chooses the variable most likely to promote extensive rewriting.

rewrites:

$$\forall \alpha, \beta : ordl.\, \alpha \cdot \beta = \alpha \cdot \beta + \alpha \cdot 0, \tag{17}$$
$$\forall \alpha, \beta : ordl.\, \alpha \cdot \beta = \alpha \cdot \beta + 0, \tag{18}$$
$$\forall \alpha, \beta : ordl.\, \alpha \cdot \beta = \alpha \cdot \beta, \tag{19}$$
$$\forall \alpha, \beta : ordl.\, true. \tag{20}$$

The redundant quantifiers are removed and the case is proved.

Case 2:

This is identical to the step case that would be produced in an attempt to proof plan the theorem using the traditional form of rippling.

$$\begin{gathered} \forall \alpha, \beta : ordl.\, \alpha.(\beta + x) = \alpha \cdot \beta + \alpha \cdot x \vdash \\ \forall \alpha, \beta : ordl.\, \alpha.(\beta + s(x)) = \alpha \cdot \beta + \alpha \cdot s(x). \end{gathered} \tag{21}$$

This is annotated and rippled as follows:

$$\forall \alpha, \beta : ordl.\, \alpha \cdot (\beta + \boxed{s(\underline{x})}^{\uparrow}) = \alpha \cdot \beta + \alpha \cdot \boxed{s(\underline{x})}^{\uparrow}, \tag{22}$$
$$\forall \alpha, \beta : ordl.\, \alpha \cdot \boxed{s(\underline{\beta + x})}^{\uparrow} = \alpha \cdot \beta + \alpha \cdot \boxed{s(\underline{x})}^{\uparrow}, \tag{23}$$
$$\forall \alpha, \beta : ordl.\, \boxed{\underline{\alpha \cdot (\beta + x)} + \alpha}^{\uparrow} = \alpha \cdot \beta + \alpha \cdot \boxed{s(\underline{x})}^{\uparrow}, \tag{24}$$
$$\forall \alpha, \beta : ordl.\, \boxed{\underline{\alpha \cdot (\beta + x)} + \alpha}^{\uparrow} = \alpha \cdot \beta + \boxed{(\underline{\alpha \cdot x} + \alpha)}^{\uparrow}. \tag{25}$$

Weak fertilisation (using the induction hypothesis as a rewrite rule) then returns the goal

$$\forall \alpha, \beta : ordl.\, (\alpha \cdot \beta + \alpha \cdot x) + \alpha = \alpha \cdot \beta + (\alpha \cdot x + \alpha). \tag{26}$$

Two generalise steps convert the goal to

$$\forall \alpha, \delta, \zeta : ordl.\, (\delta + \zeta) + \alpha = \delta + (\zeta + \alpha). \tag{27}$$

The proof then continues as a proof for the associativity of plus, using a second induction.

Case 3:

The third case is the "new" step case containing a limit ordinal.

$$\begin{gathered} \forall n : nat.\, \forall \alpha, \beta : ordl.\, \alpha \cdot (\beta + f(n)) = \alpha \cdot \beta + \alpha \cdot f(n) \vdash \\ \forall \alpha, \beta : ordl.\, \alpha \cdot (\beta + \lim(\lambda x.\ f(x)) = \alpha \cdot \beta + \alpha \cdot \lim(\lambda x.\ f(x)). \end{gathered} \tag{28}$$

The induction assumption embeds in the conclusion, giving an annotation which is shown below. The notation $\lfloor\rfloor$ shows that there is a sink in the skeleton at this point.

$$\forall \alpha, \beta : ordl.\, \alpha \cdot (\beta + \boxed{\lim(\lambda x.\ \underline{f(\lfloor x \rfloor)})}^{\uparrow} = \alpha \cdot \beta + \alpha \cdot \boxed{\lim(\lambda x.\ \underline{f(\lfloor x \rfloor)})}^{\uparrow}. \tag{29}$$

This ripples (using (11) – definition of $+$) to

$$\forall \alpha, \beta : ordl.\, \alpha \cdot \boxed{\lim(\lambda x.\ \underline{\beta + f(\lfloor x \rfloor)})}^{\uparrow} = \alpha \cdot \beta + \alpha \cdot \boxed{\lim(\lambda x.\ \underline{f(\lfloor x \rfloor)})}^{\uparrow}. \quad (30)$$

The proof then proceeds rewriting with the definition of multiplication,

$$\forall \alpha, \beta : ordl.\, \boxed{\lim(\lambda x.\ \underline{\alpha \cdot (\beta + f(\lfloor x \rfloor))})}^{\uparrow} = \alpha \cdot \beta + \alpha \cdot \boxed{\lim(\lambda x.\ \underline{f(\lfloor x \rfloor)})}^{\uparrow}. \quad (31)$$

$\lambda Clam$ then rewrites using the definition of multiplication,

$$\forall \alpha, \beta : ordl.\, \boxed{\lim(\lambda x.\ \underline{\alpha \cdot (\beta + f(\lfloor x \rfloor))})}^{\uparrow} = \alpha \cdot \beta + \boxed{\lim(\lambda x.\ \underline{\alpha \cdot f(\lfloor x \rfloor)})}^{\uparrow} \quad (32)$$

$\lambda Clam$ then rewrites using the definition of addition,

$$\forall \alpha, \beta : ordl.\, \boxed{\lim(\lambda x.\ \underline{\alpha \cdot (\beta + f(\lfloor x \rfloor))})}^{\uparrow} = \boxed{\lim(\lambda x.\ \underline{\alpha \cdot \beta + \alpha \cdot f(\lfloor x \rfloor)})}^{\uparrow} \quad (33)$$

$\lambda Clam$ then rewrites using (12),

$$\forall \alpha, \beta : ordl.\, \boxed{\forall n : nat.\, \underline{\alpha \cdot (\beta + f(\lfloor n \rfloor)) = \alpha \cdot \beta + \alpha \cdot f(\lfloor n \rfloor)}}^{\uparrow}. \quad (34)$$

Notice here that the formula has been β-reduced implicitly. $\lambda Clam$ then rewrites using (13) twice,

$$\forall \alpha : ordl.\, \boxed{\forall n : nat.\, \underline{\forall \beta : ordl.\, \alpha \cdot (\beta + f(\lfloor n \rfloor)) = \alpha \cdot \beta + \alpha \cdot f(\lfloor n \rfloor)}}^{\uparrow} \quad (35)$$

$$\forall n : nat.\, \forall \alpha, \beta : ordl.\, \alpha \cdot (\beta + f(\lfloor n \rfloor)) = \alpha \cdot \beta + \alpha \cdot f(\lfloor n \rfloor) \quad (36)$$

This then succeeds by strong fertilisation (i.e. the conclusion is now identical to the induction hypothesis). It is no coincidence that the hypothesis and conclusion match at this point. The rippling heuristic is explicitly designed to bring this about. This whole proof has been automated in $\lambda Clam$.

5.2 The Synthesis of a Fixpoint for a Normal Function

One of our primary interests in creating $\lambda Clam$ was to investigate the use of least-commitment reasoning in order to perform synthesis proofs. By this we mean placing uninstantiated meta-variables in proof goals which become instantiated during the course of the proof and generate an existential witness. We looked at a simple case where this style of reasoning could be using in the ordinal domain. This synthesis proof was performed automatically in $\lambda Clam$. It made no use of induction succeeding with the symbolic evaluation method alone.

Definition 1. *A function, ϕ, from ordinals to ordinals that is strictly increasing (preserves the order) and continuous is called* normal.

Theorem 2. *Any normal ϕ has a fixed point:*

$$\exists\eta.\ \phi(\eta) = \eta. \tag{37}$$

The continuity of an increasing function F is expressed by the rewrite

$$\phi(\lim(\lambda z.\ F(z))) \Leftarrow:\Rightarrow \lim \lambda z.\ \phi(F(z)). \tag{38}$$

We were able to synthesize such a fixed point using symbolic evaluation and the lemmas:

$$\lim \lambda n.\ F(s(n)) \Leftarrow:\Rightarrow \lim \lambda n.\ F(n), \tag{39}$$
$$\lambda n.\ F(F^n(A)) \Leftarrow:\Rightarrow \lambda n.\ F^{s(n)}(A) \tag{40}$$

where the second is simply a defining property of the iteration of function application.

The final planning proceeds as follows. Here, capital letters indicate meta-variables whose instantiation is refined during the planning process.

$$\begin{array}{rcll}
\exists\eta.\ \phi(\eta) & = & \eta & \\
\phi(E) & = & E & \text{Exists Introduction} \\
\lim \lambda z.\ \phi(F(z)) & = & \lim \lambda z.\ F(z) & \text{Rewriting LHS with (38)} \\
\lim \lambda z.\ \phi^{s(z)}(A) & = & \lim \lambda z.\ \phi^{z}(A) & \text{Rewriting LHS with (40)} \\
\lim \lambda z.\ \phi^{z}(A) & = & \lim \lambda z.\ \phi^{z}(A) & \text{Rewriting LHS with (39)} \\
 & \mathit{true}. & & \text{Rewriting the identity}
\end{array}$$

As can be seen the system gradually instantiates the meta-variable, E, as it rewrites the terms. The plan shows that $\lim \lambda z.\ \phi^{z}(A)$ is a fixed point; as A is a meta-variable here, this shows that the proof will hold for *any* instantiation of A.

This plan is found with the small but appropriately chosen set of definitions and lemmas shown in this section. We feel it demonstrates the possible applications for higher-order proof planning systems in the ordinal domain.

6 Related and Further Work

Different mechanisations of reasoning about ordinals and cardinals have been carried out previously. For example, Paulson [17] introduces ordinals in the course of a development of set theory. While providing the foundational assurance of a development from first principles, this work assumes a fair amount of user interaction in building up proofs. A further development in this style is in [18]. Closer in spirit to our presentation is the introduction of ordinals in the Coq system, though again user guidance is assumed in building proofs as a new datatype.

The system, ACL2, described by Boyer & Moore [3] makes use of induction over ordinals to strengthen the proof system. However, this feature is hidden in the system's "black box" treatment used to check termination of user-defined functions, and is not directly accessible by the user. PVS [16] also contains a construction of the ordinals up to ϵ_0 and a well-foundedness proof for the associated order based on the development in ACL2. *check this – is the well-foundedness simply asserted?*

There are a number of extensions we would like to make to this case study. At a basic level we should like to include definitions for sums and products as presented by Suppes [21] and attempt to plan theorems with them. We should also like to implement the work on transitivity which allowed proofs involving order relations to be planned in *Clam* in λ*Clam* in order to plan theorems about the order on ordinals.

Moving on from this there are a number of more challenging theorems (as illustrated by our trial synthesis of a fixed point for a continuous function) to which our theory could be extended and which we believe would be a fruitful area of study for mechanized higher order proof.

7 Conclusion

We have chosen to examine ordinal arithmetic as a test case for the extension of the ideas of rippling and proof planning in a higher order setting. Our aim was to maintain the natural higher-order presentation of the ordinals while retaining full automation of the proof generation process.

We were greatly helped in our case study by the higher order features built into λProlog which removed the need for explicitly stating β-reduction as a rewrite rule and allowed side-conditions preventing variable capture to be left to the underlying language to handle.

With a small amount of theory building we were able to successfully plan standard undergraduate textbook examples and exercises. We were also able to demonstrate how proof planning, in particular least commitment devices such as the use of meta-variables, could have a part to play in the mechanisation of proof theoretic results in the ordinal domain. These results were confirmation that λ*Clam* can naturally handle higher order examples and that λProlog is a suitable language in which to express such mathematical concepts.

Thus, we claim to have identified common patterns of proof search that extend automated inductive theorem proving to the mathematically interesting higher order domain of ordinals and ordinal functions.

Acknowledgements

This research was funded by EPSRC grant Gr/m45030. We thank members of the Mathematical Reasoning Group, and especially Alan Bundy for input to this work. The presentation has benefited from the comments of the anonymous referees.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.
3. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Perspectives in Computing, Vol 23.
4. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.
5. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.
6. Alan Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier., 1998. Forthcoming.
7. A. Cichon and H. Touzet. An ordinal calculus for proving termination in term rewriting. In *Proceedings of CAAP'96, Coll. on Trees in Algebra and Programming*, number 1059 in Lecture Notes in Computer Science. Springer, 1996.
8. Th. Coquand. *Une Théorie des Constructions.* PhD thesis, University of Paris VII, 1985.
9. Herbert B. Enderton. *Elements of Set Theory.* Academic Press, 1977.
10. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS : an interactive mathematical proof system. *Journal of Automated Reasoning*, 9(11):213–248, 1993.
11. A. Felty. A logic programming approach to implementing higher-order term rewriting. In L-H Eriksson et al., editors, *Second International Workshop on Extensions to Logic Programming*, volume 596 of *Lecture Notes in Artificial Intelligence*, pages 135–61. Springer-Verlag, 1992.
12. P. Halmos. *Naive Set Theory.* Van Nostrand, Princeton, NJ, 1960.
13. A. G. Hamilton. *Numbers, sets and axioms: the apparatus of mathematics.* Cambridge University Press, 1982.
14. Jean-Pierre Jouannaud and Mitsuhiro Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In *Proceedings of the Eighteenth EATCS Colloquium on Automata, Languages and Programming*, Madrid, Spain, 1991.
15. Z. Luo and R. Pollack. Lego proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, May 1992.
16. S. Owre, J. M. Rushby, and N. Shankar. PVS : An integrated approach to specification and verification. Tech report, SRI International, 1992.
17. L.C. Paulson. Set theory for verification: II. induction and recursion. *Journal of Automated Reasoning*, 15:353–389, 1995.
18. L.C. Paulson and K. Grabczewski. Mechanizing set theory: cardinal arithmetic and the axiom of choice. *Journal of Automated Reasoning*, pages 291–323, 1996.
19. J.D.C Richardson, A. Smaill, and Ian Green. System description: proof planning in higher-order logic with LambdaCLAM. In Claude Kirchner and Hélène Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 129–133, Lindau, Germany, July 1998.

20. Alan Smaill and Ian Green. Higher-order annotated terms for proof search. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1275 of *Lecture Notes in Computer Science*, pages 399–414, Turku, Finland, 1996. Springer-Verlag. Also available as DAI Research Paper 799.
21. P. Suppes. *Axiomatic Set Theory.* Van Nostrand, Princeton, NJ, 1960.