Division of Informatics, University of Edinburgh

Institute for Representation and Reasoning

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

by

Alan Smaill, Louise Dennis

Informatics Research Report EDI-INF-RR-0040

Division of Informatics **April 2001**
http://www.informatics.ed.ac.uk/

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

Alan Smaill, Louise Dennis

Informatics Research Report EDI-INF-RR-0040

DIVISION *of* INFORMATICS
Institute for Representation and Reasoning

April 2001

Submitted to TpHOLs 2001

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 the proof planning inductive of 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 LambdaClam, a higher order proof planning system for induction, and standard undergraduate text book problems have been successfully planned with a simple extension to the standard machinery for higher order rippling and induction. We show the synthesis of a fixpoint for normal ordinal functions which demonstrates how our automation could be extended to produce more interesting results that the textbook examples tried so far.

Keywords :

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

Alan Smaill** and Louise A. Dennis

Division of Informatics, University of Edinburgh, 80 South Bridge, Edinburgh, EH1 1HN, UK, A.Smaill@ed.ac.uk, louised@dai.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 the proof planning inductive of 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 $\lambda Clam$, a higher order proof planning system for induction, and standard undergraduate text book problems have been successfully planned with a simple extension to the standard machinery for higher order rippling and induction. We show the synthesis of a fixpoint for normal ordinal functions which demonstrates how our automation could be extended to produce more interesting results that the textbook examples tried so far.

1 Introduction

This paper reports on using $\lambda Clam$ to plan proofs about ordinal arithmetic, making use of higher order features. A standard text book presentation of ordinal numbers [17] was chosen to provide a basis for a theory of ordinal arithmetic and the system was used to attempt to plan proofs of examples appearing in a number of text books [17, 7, 11] 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 only minor 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 research was funded by EPSRC grant Gr/m45030

** Corresponding Author

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 $\lambda Clam$ (§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.

Ordinal arithmetic is of interest, among other things, for its use in aiding termination proofs (see [6]), and in classifying proof-theoretic complexity.

Our presentation in this section follows the standard exposition from set theory; however, our implemented formalisation does not attempt to build definitions from first principles in this way, but adopts various assumptions as a starting point. We indicate these assumptions later in the paper.

Ordinals can be thought of as (equivalence classes of) well-ordered linear orders. As such they are a generalisation of the usual natural numbers, but the arithmetic extends beyond the finite orders. Each ordinal has all smaller ordinals as members and the membership relation provides the appropriate well-ordering.

Definitions (from [17]) follow:

Definition 1.

$$\epsilon A = \{\langle x, y\rangle : x \in A \wedge y \in A \wedge x \in y\}. \tag{1}$$

Definition 2. *A is* complete *if and only if every member of A is a subset of A.*

Definition 3. *A is an* ordinal *if and only if A is complete and ϵA well-orders A*

Definition 4.

$$\forall \alpha. s(\alpha) = \alpha \cup \{\alpha\}. \tag{2}$$

From this we derive standard constructions of the natural numbers (i.e. $0 = \emptyset$, $1 = \{\emptyset\}$, $2 = \{\emptyset, \{\emptyset\}\}$ etc.). The natural numbers form the finite ordinals, but it is also possible to have infinite ordinals.

Theorem 1. $\forall \alpha : ordl. \bigcup \alpha$ *is an ordinal*

We introduce here the type of ordinals, *ordl*; as is well known, this cannot correspond to a set.

Definition 5. *A* limit ordinal *is a non-zero ordinal which has no immediate predecessor.*

Given infinite sets, theorem 1 implies the existence of limit ordinals. Among other things, the set ω of all natural numbers with the usual order is a limit ordinal.

These observations allow us to develop principles of transfinite induction over ordinals (of which the following is one of many).

Theorem 2 (Principle of Transfinite Induction). *Suppose that*

1. $\phi(0)$;
2. *for every* α, *if* $\phi(\alpha)$ *then* $\phi(s(\alpha))$;
3. *for every limit ordinal* γ, *if for every* $\beta < \gamma$, $\phi(\beta)$ *then* $\phi(\gamma)$.

Then for every ordinal $\alpha.\phi(\alpha)$.

This makes ordinals a credible test case for inductive theorem provers. Operations over limit ordinals are frequently presented as

$$\cup_{\beta<\alpha} F(\beta),$$

for some F of functional type. This makes such proofs naturally higher order.

A standard way of defining functions over the ordinals is using three cases rather than the two for natural numbers.

$$F(0) = a, \tag{3}$$
$$F(s(x)) = G(x, F(x)), \tag{4}$$
$$limit(\alpha) \rightarrow F(\cup_{\beta<\alpha}\beta) = H(\alpha, F \restriction \{\beta \ : \ \beta < \alpha\}), \tag{5}$$

where $\restriction$ is restriction of a function to a specified domain.

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

$$x + 0 = x, \tag{6}$$
$$x + s(y) = s(x + y), \tag{7}$$
$$limit(\alpha) \rightarrow x + \cup_{\beta<\alpha} F(\beta) = \cup_{\beta<\alpha}(x + F(\beta)). \tag{8}$$

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. There is a good account of the ideas involved in [10].

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 [3]. A proof plan is a proof of a theorem, at some level of abstraction, presented as a tree. Traditionally, each node in this tree is justified by a tactic. The nature of these tactics varies from system to system. They may be sequences of inference rules, programs for generating sequences of inferences or a further proof plan at some lower level of abstraction.

In principle, while the generation of the proof tree may have involved heuristics and (possibly) unsound inference steps, it can be justified by executing the tactics attached to the nodes. In practice the generation of the proof tree often involves the execution of the tactics subsequently used to justify the nodes and so the distinction between methods and tactics can become blurred.

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. Proof methods are therefore partial tactic specifications presented in a pre- and postcondition fashion. In theory the method's pre- and postconditions describe (partially) in some meta-language the proof state before and after the application of their associated tactic.

The first and most famous proof plan is that for induction with the associated rippling heuristic. This was implemented in the *Clam* proof planner. λ*Clam* [15] 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. A plan is formed using methods placed within a *method waterfall*. The waterfall determines a general strategy for the process of the proof and imposes a hierarchy upon the methods. This means that only certain methods are considered as viable ways to extend the plan at certain points in a proof. There are two sorts of method; compound and atomic. Compound methods determine the structure of the waterfall. A compound method is a *methodical expression* built from methods. *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.

[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.

$\lambda Clam$ [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 Plan for Induction and Rippling

Induction is the main proof strategy implemented in $\lambda Clam$. It is defined by a compound method called `induction_top_meth`. The body of this is defined as follows:

```
(repeat_meth
        (orelse_meth trivial
        (orelse_meth allFi
        (orelse_meth taut
        (orelse_meth sym_eval
        (orelse_meth (repeat_meth generalise)
                     ind_strat
))))))
```

This is a methodical expression and defines the strategy which is to repeatedly apply one of `trivial` (looking for goals that have become true), `allFi` (removing higher order universal quantifiers), `taut` (a tautology checker), `sym_eval` (symbolic evaluation), `repeat_meth generalise` (generalise away as many common subterms as possible) and `ind_strat` (the induction strategy proper).

`ind_strat` is itself a compound method. It applies the `induction_meth` method which performs ripple analysis [5] to choose an induction scheme (from a selection specified in $\lambda Clam$'s theories) and produces subgoals for base and step cases. `ind_strat` leaves the base cases unaltered and so they are tackled by the next repeat of the `induction_top_meth` methodical expression. `ind_strat` applies the `step_case` method to the remaining cases.

The `step_case` method is again compound and uses four submethods in sequence, `set_up_ripple` (find an initial embedding) then `repeat_meth wave_method` (ripple as much as possible), `post_ripple` (remove the embedding) and `fertilise` (exploit the induction hypothesis to solve or rewrite the conclusion). At this point the resulting goal feeds back into the `induction_top_meth` expression which can perform additional simplification or further inductions if required.

The `wave_method` method embodies the rippling heuristic. Rippling was first introduced in [4]. We use the theory as presented in [16]. Rippling steps apply rewrite rules to a target term which is annotated 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 [9]. There

[2] Details of $\lambda Clam$ can be found at `http://dream.dai.ed.ac.uk/systems/lambda-clam/`. The Teyjus implementation can be obtained by emailing `dream@dai.ed.ac.uk`.

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 rippling that used annotated *wave rules* to rewrite terms.

Rippling is terminating [1]. 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 1. The method will backtrack in order to try to satisfy all

---

Input

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

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

Preconditions

1. The conditional rewrite rule *Rule*, $Cond \rightarrow X :\Rightarrow Y$, applies to G and rewrites it to G'.
2. There is an embedding E' that embeds S in G'.
3. $E' <_\mu E$.
4. $Cond = True$ or $Cond \in H$.

Output

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

Fig. 1. The Wave Method

requirements, and if it is successful returns a new goal.

Embeddings Embeddings are described by a tree data structure. Embedding trees describe how a skeleton embeds in a term. 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 term into which the skeleton is to be embedded. A leaf node in an embedding tree indicates that the term trees of the skeleton and the target term match below this point. A node in an embedding tree will appear at a function application node in the skeleton term tree and indicates that this node is matched to the function application term in the target term tree at the indicated address.

Example 1. Consider embedding the term $x+y$ into the term $s(x)+y$. We do this as in figure 2. The two terms are shown as trees with branches represented by solid lines. The address of each node is given. 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.

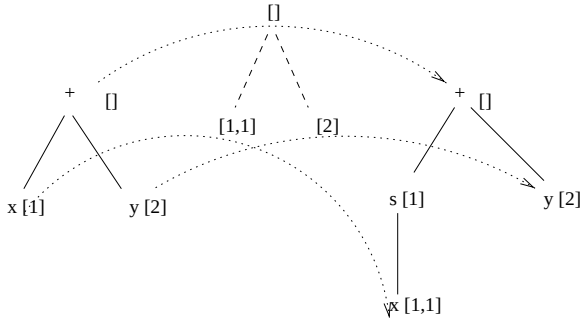

Fig. 2. An Embedding

The embedding tree for this would be (node [ ] [(leaf [1,1]) (leaf [2])])). This states that the function application at the top of $x+y$ (i.e. $+$) matches with the node at address [ ] of $s(x)+y$ (i.e. $+$), that the argument x matches the sub term at [1,1] (i.e. the s has been skipped over and x matches x) and so on.

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

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

NB. It is important to remember that the annotations do not actually exist in the implementation which records instead the skeleton and embedding[3]. These annotations are just a presentational convenience.

The whole proof strategy for induction can be seen in figure 3.

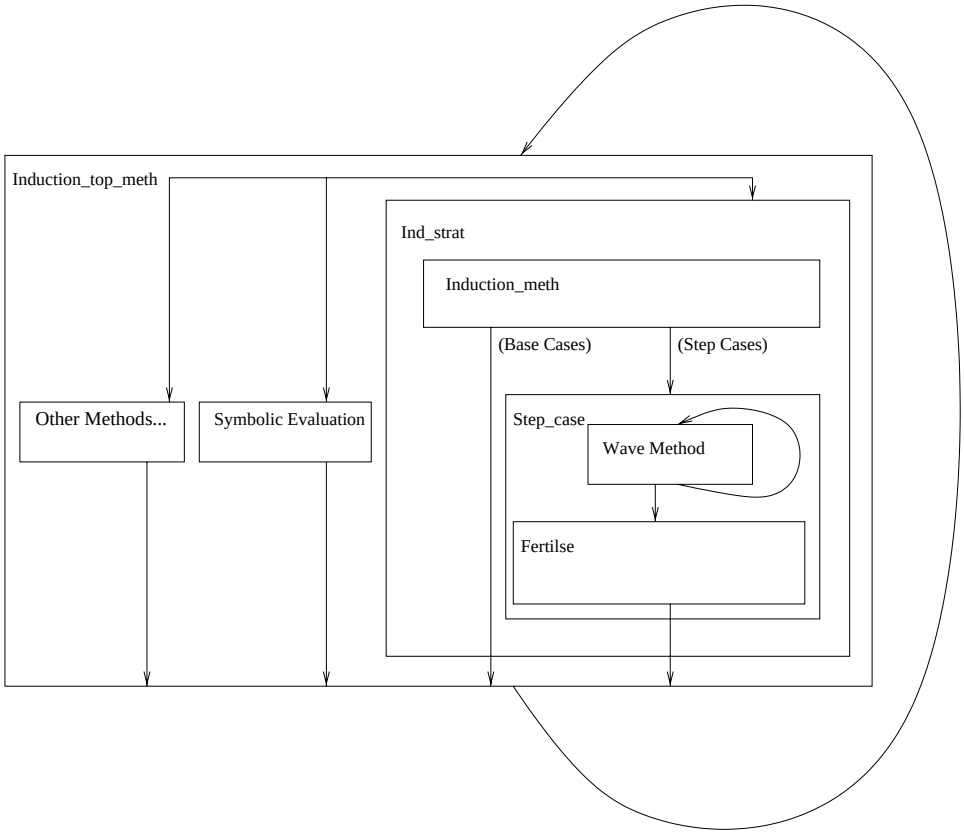

Fig. 3. The Proof Strategy for Induction

4 Proof Planning Ordinal Arithmetic

To implement ordinal arithmetic in $\lambda Clam$ we faced a number of tasks.

[3] In fact $\lambda Clam$ 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.

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

As part of this last task we found we had to extend our idea of the skeleton of an embedding.

4.1 Representing Ordinals in $\lambda Clam$

It is simple to supply new constants and types for $\lambda Clam$ 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, or taking the union (limit) of a set of ordinals. We overloaded the types of the 0 and s functions provided in $\lambda Clam$'s arithmetic theory[4] and so were able to use these as constructors for a type *ordl*.

We wanted a "common form" for limit ordinals so that we could exploit pattern matching. The usual presentation of functions defined at limit ordinals is as a union:

$$\cup_{\beta<\alpha} F\beta.$$

We chose to use the notation lim (the "limit" of the function F) rather than $\cup$ in the implementation. In fact, the definitional rewrite rules for plus etc. express the continuity of the standard operations, with respect to the initial topology; however, the reader should understand uses of lim in what follows to correspond to the traditional $\cup$.

We use $\lim_\alpha F$ to represent $\lim_{\beta<\alpha} F(\beta)$. In particular we used

$$\alpha = \lim_{\alpha} \lambda x.x$$

for limit ordinals. This is the defining property for limit ordinals[5] and it follows that such ordinals are not of the form $s(x)$.

4.2 Defining the Arithmetical Operators

The first two clauses in the definition of plus already existed in the arithmetic theory and were "borrowed" from there. NB. that + had been defined with recursion on the second variable as opposed to the first (which is more usual). This was in anticipation of the case study. $\lambda Clam$ allows a user to state exactly which lemmas and definitions they wish to use and does not type check except where absolutely necessary allowing rewrite rules to be reused in this way wherever

[4] In principle we could have provided new constants instead. This approach allowed us to "borrow" appropriate existing rewrite rules from the arithmetic theory (provided they still held for ordinal arithmetic, of course!).

[5] 0 is not a limit ordinal however so $\alpha \neq 0$ is added as a hypothesis wherever necessary.

they are applicable. This does place a burden on the user, however, to use only a consistent set of rewrites and to assume as minimal a set of lemmas as possible (more of this in §4.3).

The statement of continuity of addition is as follows,where F maps ordinals to ordinals,

$$x + \lim_{\beta<\alpha} F(\beta) = \lim_{\beta<\alpha}(x + F(\beta)).$$

This then yields the rewrite rule

$$X + \lim_{A} F :\Rightarrow \lim_{A} \lambda y.(x + F(y))$$

in $\lambda Clam$. 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. $(s(X) = s(Y)) :\Rightarrow (X = Y)$) and identity ($(X = X) :\Rightarrow true$) existing in $\lambda Clam$. We also introduced the following new rewrites that correspond to basic properties of ordinal functions.

$$\lim_{\alpha} F = \lim_{\alpha} G :\Rightarrow (\forall o : ordl.o < \alpha \Rightarrow (Fo) = (Go)), \tag{9}$$

$$\alpha \neq 0 \Rightarrow (\lim_{\alpha} F = C :\Rightarrow (\forall o : ordl.o < \alpha \Rightarrow (Fo) = C)), \tag{10}$$

$$\lim_{s(\alpha)} \lambda y.y :\Rightarrow \alpha. \tag{11}$$

These rewrite rules are obliged to observe polarity constraints to ensure soundness.

Finally, we use the following logical equivalence, which gives rise to two rewrites:

$$(\forall x : \tau_1.\forall y : \tau_2.\Phi(y) \Rightarrow \Psi(x, y)) \leftrightarrow (\forall y : \tau_2.\Phi(y) \Rightarrow \forall x : \tau_1.\Psi(x, y)). \tag{12}$$

4.4 An Induction Scheme for Transfinite Induction

The induction scheme used is standard for ordinals, though the condition that the variable is a limit ordinal in the "new" step case is omitted. Instead we use pattern matching with our representation of limit ordinals. The new step case, for an initial goal of the form $\forall x : ordl.P(x)$ is

$$\forall \beta : ordl.\beta < \alpha \rightarrow P(\beta) \vdash P(\lim_{\alpha} \lambda x.x). \tag{13}$$

The choice of initial embedding here raises some problems. The goal is not syntactically an expansion of the hypothesis. There is no implication or quantifier in the conclusion. We adapted the notion of a *wild card* symbol, $*$, that had been used in experiments with course of values induction. This allowed us to embed

$P(*)$ in $P(\lim_\alpha \lambda x.x)$ by matching $*$ to $\lim_\alpha \lambda x.x$. We also added a new case to `set_up_ripple`. In the standard presentation `set_up_ripple` adopts the whole induction hypothesis as the skeleton. In our new case it picked $P(*)$ as the skeleton dropping the condition $\beta < \alpha$ and replacing the universally quantified β with the wild card.

All this was done in a specialised ordinals module. The basic machinery for rippling and induction was carried over, once the matching condition above had been supplied.

5 Evaluation

This machinery was enough to plan automatically the following goals.

$$\forall \alpha : ordl.0 + \alpha = \alpha, \tag{14}$$
$$\forall \alpha : ordl.\alpha + 1 = s(\alpha), \tag{15}$$
$$\forall \alpha, \beta, \gamma : ordl.(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma), \tag{16}$$
$$\forall \alpha : ordl.0.\alpha = 0, \tag{17}$$
$$\forall \alpha : ordl.1.\alpha = \alpha, \tag{18}$$
$$\forall \alpha : ordl.\alpha.1 = \alpha, \tag{19}$$
$$\forall \alpha : ordl.\alpha.2 = \alpha + \alpha, \tag{20}$$
$$\forall \alpha, \beta, \gamma : ordl.\alpha.(\beta + \gamma) = \alpha.\beta + \alpha.\gamma, \tag{21}$$
$$\forall \alpha, \beta, \gamma : ordl.(\alpha.\beta).\gamma = \alpha.(\beta.\gamma), \tag{22}$$
$$\forall \alpha : ordl.\alpha^1 = \alpha, \tag{23}$$
$$\forall \alpha : ordl.1^\alpha = 1, \tag{24}$$
$$\forall \alpha : ordl.\alpha^2 = \alpha.\alpha, \tag{25}$$
$$\forall \alpha, \beta, \gamma : ordl.\alpha^{\beta+\gamma} = \alpha^\beta.\alpha^\gamma, \tag{26}$$
$$\forall \alpha, \beta, \gamma : ordl.\alpha^{\beta.\gamma} = \alpha^{\beta^\gamma}. \tag{27}$$

These were drawn from exercises in three undergraduate text books [17, 7, 11]. They represent all the 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.

5.1 The Distributivity of Multiplication over Addition

As a sample of the sort of plan produced by $\lambda Clam$ 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:**

$\lambda Clam$ performs backwards proof so it starts with the initial goal.

$$\forall \alpha, \beta, \gamma : ordl.\alpha.(\beta + \gamma) = \alpha.\beta + \alpha.\gamma. \tag{28}$$

`induction_top_meth` is applied to this. Only one of its submethods succeeds, `ind_strat`, because its submethod, `induction_meth` succeeds. Ripple analysis[6] suggests the use of induction on γ. This splits the goal into three subgoals; one base case and two step cases.

Case 1:

$$\forall \alpha, \beta : ordl.\alpha.(\beta + 0) = \alpha.\beta + \alpha.0. \tag{29}$$

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 rewrites:

$$\forall \alpha, \beta : ordl.\alpha.\beta = \alpha.\beta + \alpha.0, \tag{30}$$
$$\forall \alpha, \beta : ordl.\alpha.\beta = \alpha.\beta + 0, \tag{31}$$
$$\forall \alpha, \beta : ordl.\alpha.\beta = \alpha.\beta, \tag{32}$$
$$\forall \alpha, \beta : ordl.T. \tag{33}$$

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 in standard arithmetic.

$$\begin{array}{c} \forall \alpha, \beta : ordl.\alpha.(\beta + x) = \alpha.\beta + \alpha.x \vdash \\ \forall \alpha, \beta : ordl.\alpha.(\beta + s(x)) = \alpha.\beta + \alpha.s(x). \end{array} \tag{34}$$

This is annotated and rippled as follows:

$$\forall \alpha, \beta : ordl.\alpha.(\beta + \boxed{s(\underline{x})}^{\uparrow}) = \alpha.\beta + \alpha.\boxed{s(\underline{x})}^{\uparrow}, \tag{35}$$

$$\forall \alpha, \beta : ordl.\alpha.\boxed{s(\underline{\beta + x})}^{\uparrow} = \alpha.\beta + \alpha.\boxed{s(\underline{x})}^{\uparrow}, \tag{36}$$

$$\forall \alpha, \beta : ordl.\boxed{\underline{\alpha.(\beta + x)} + \alpha}^{\uparrow} = \alpha.\beta + \alpha.\boxed{s(\underline{x})}^{\uparrow}, \tag{37}$$

$$\forall \alpha, \beta : ordl.\boxed{\underline{\alpha.(\beta + x)} + \alpha}^{\uparrow} = \alpha.\beta + \boxed{(\underline{\alpha.x} + \alpha)}^{\uparrow}. \tag{38}$$

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

$$\forall \alpha, \beta : ordl.(\alpha.\beta + \alpha.\gamma) + \alpha = \alpha.\beta + (\alpha.\gamma + \alpha). \tag{39}$$

Two generalise steps convert the goal to

$$\forall \alpha, \delta, \zeta : ordl.(\delta + \zeta) + \alpha = \delta + (\zeta + \alpha). \tag{40}$$

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

The proof then proceeds with a second induction as for the associativity of plus.

Case 3:

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

$$\begin{array}{c}\forall\delta : ordl.(\delta < \gamma) \Rightarrow \forall\alpha, \beta : ordl.\alpha.(\beta + \delta) = \alpha.\beta + \alpha.\delta \vdash \\ \forall\alpha, \beta : ordl.\alpha.(\beta + \lim_\gamma \lambda x.x) = \alpha.\beta + \alpha.\lim_\gamma \lambda x.x.\end{array} \tag{41}$$

This is then annotated using $\alpha.(\beta + *)$ as the skeleton. We will use the notation $\lfloor x \rfloor$ to indicate when a subterm has matched the wild card[7]. In this way the above goal annotates and ripples as follows:

$$\forall\alpha, \beta : ordl.\alpha.(\beta + \lfloor \lim_\gamma \lambda x.x \rfloor) = \alpha.\beta + \alpha.\lfloor \lim_\gamma \lambda x.x \rfloor. \tag{42}$$

There are no wave fronts initially only a matching of the wild card. This ripples to

$$\forall\alpha, \beta : ordl.\alpha.\boxed{\lim_\gamma \lambda x.\underline{(\beta + \lfloor x \rfloor)}}^{\uparrow} = \alpha.\beta + \alpha.\lfloor \lim_\gamma \lambda x.x \rfloor. \tag{43}$$

Annotations have now appeared in the term this is because the rewrite rule has introduced new structure and this is reflected in the embedding that was found. Note here that we have applied β-reduction in the course of the ripple. λProlog handles this automatically in the preconditions for the rippling method and no additional β-reduction rewrite rules had to be supplied.

$$\forall\alpha, \beta : ordl.\boxed{\lim_\gamma \lambda x.\underline{\alpha.(\beta + \lfloor x \rfloor)}}^{\uparrow} = \alpha.\beta + \alpha.\lfloor \lim_\gamma \lambda x.x \rfloor, \tag{44}$$

$$\forall\alpha, \beta : ordl.\boxed{\lim_\gamma \lambda x.\underline{\alpha.(\beta + \lfloor x \rfloor)}}^{\uparrow} = \alpha.\beta + \boxed{\lim_\gamma \lambda x.\underline{\alpha.\lfloor x \rfloor}}^{\uparrow}, \tag{45}$$

$$\forall\alpha, \beta : ordl.\boxed{\lim_\gamma \lambda x.\underline{\alpha.(\beta + \lfloor x \rfloor)}}^{\uparrow} = \boxed{\lim_\gamma \lambda x.\underline{\alpha.\beta + \alpha.\lfloor x \rfloor}}^{\uparrow}, \tag{46}$$

$$\forall\alpha, \beta, \boxed{\delta : ordl.\delta < \gamma \Rightarrow \underline{\alpha.(\beta + \lfloor \delta \rfloor) = \alpha.\beta + \alpha.\lfloor \delta \rfloor}}^{\uparrow}, \tag{47}$$

$$\forall\alpha, \boxed{\delta : ordl.\delta < \gamma \Rightarrow \underline{\forall\beta : ordl.\alpha.(\beta + \lfloor \delta \rfloor) = \alpha.\beta + \alpha.\lfloor \delta \rfloor}}^{\uparrow}, \tag{48}$$

$$\boxed{\forall\delta : ordl.\delta < \gamma \Rightarrow \underline{\forall\alpha, \beta : ordl.\alpha.(\beta + \lfloor \delta \rfloor) = \alpha.\beta + \alpha.\lfloor \delta \rfloor}}^{\uparrow}. \tag{49}$$

This then succeeds by strong fertilisation (i.e. the conclusion is now identical to the induction hypothesis).

[7] For those familiar with rippling this should not be confused with a sink annotation although its function is similar.

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 made no use of induction.

Definition 6. *A function, ϕ, from ordinals to ordinals that is strictly increasing (preserves the order) and continuous is called* normal.

Theorem 3. *Any normal ϕ has a fixed point.*

$$\exists \eta.\phi(\eta) = \eta \tag{50}$$

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

$$\phi(\lim_X(\lambda z.F(z))) :\Rightarrow \lim_X \lambda z.\phi(F(z)). \tag{51}$$

(The corresponding implication is true even when X is not a limit ordinal.)

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

$$\lim_\omega \lambda n.F(s(n)) :\Rightarrow \lim_\omega \lambda n.F(n), \tag{52}$$

$$\lambda n.F(F^n(A)) :\Rightarrow \lambda n.F^{s(n)}(A) \tag{53}$$

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.

$$\exists \eta.\phi(\eta) = \eta, \tag{54}$$

$$\phi(E) = E, \tag{55}$$

$$\lim_X \lambda z.\phi(F(z)) = \lim_X \lambda z.F(z), \tag{56}$$

$$\lim_X \lambda z.\phi^{s(z)}(A) = \lim_X \lambda z.\phi^z(A), \tag{57}$$

$$\lim_\omega \lambda z.\phi^z(A) = \lim_\omega \lambda z.\phi^z(A), \tag{58}$$

$$T. \tag{59}$$

As can be seen the system gradually instantiates the meta-variable, E, as it rewrites the terms. The plan shows that $\lim_\omega \lambda z.\phi^z(A)$ 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 a small but appropriately chosen set of definitions and lemmas available. 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, [13] 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 [14]. 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 described in [2] 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[12] 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.

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 in [17] and attempt to plan theorems with them. We should also like to port the work on transitivity from *Clam* to 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.

We would also like to generalise our notion of a wild card (possibly by re-examining examples using course of values induction) so that instead of being a special case for handling the ordinal induction scheme it became a general addition to the theory of rippling.

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 had to extend our notion of embedding to include skeletons that contained wild cards that could match any term. We anticipate that this extension may prove to have wider applicability. 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 and the above mentioned extension to rippling in place 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 devises 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 $\lambda Clam$ 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.

References

1. David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.
2. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988. Perspectives in Computing, Vol 23.
3. 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.
4. 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.
5. Alan Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier., 1998. Forthcoming.
6. A. Cichon and H. Touzet. An ordinal calculus for proving termination in term rewriting. In Springer, editor, *Proceedings of CAAP'96, Coll. on Trees in Algebra and Programming*, number 1059 in Lecture Notes in Computer Science, 1996.
7. Herbert B. Enderton. *Elements of Set Theory*. Academic Press, 1977.
8. 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.
9. 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.
10. P. Halmos. *Naive Set Theory*. Van Nostrand, Princeton, NJ, 1960.
11. A. G. Hamilton. *Numbers, sets and axioms: the apparatus of mathematics*. Cambridge University Press, 1982.
12. S. Owre, J. M. Rushby, and N. Shankar. PVS : An integrated approach to specification and verification. Tech report, SRI International, 1992.
13. L.C. Paulson. Set theory for verification: II. induction and recursion. *Journal of Automated Reasoning*, 15:353–389, 1995.
14. L.C. Paulson and K. Grabczewski. Mechanizing set theory: cardinal arithmetic and the axiom of choice. *Journal of Automated Reasoning*, pages 291–323, 1996.
15. 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.

16. 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.
17. P. Suppes. *Axiomatic Set Theory*. Van Nostrand, Princeton, NJ, 1960.