School of Computer Science and Information Technology
University of Nottingham
Jubilee Campus
NOTTINGHAM NG8 1BB, UK

Computer Science Technical Report No. NOTTCS-WP-2005-1

Embeddings as a Higher-Order Representation of Annotations for Rippling

L. A. Dennis, I. Green and A. Smaill

First released: 2005

Embeddings as a Higher-Order Representation of Annotations for Rippling

L. A. Dennis (lad@cs.nott.ac.uk)
School of Computer Science and Information Technology, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham, UK, NG8 1BB

I. Green (ian.green@telelogic.com)
Telelogic Technologies UK Ltd, Edinburgh Laboratories, Hanover Buildings, Rose Street, Edinburgh, EH2 2NN

A. Smaill (a.smail@ed.ac.uk)
Centre for Intelligent Systems and their Applications, Division of Informatics, University of Edinburgh, Appleton Tower, Edinburgh, UK

Abstract. A notion of *embedding* appropriate to higher-order syntax is described. This provides a representation of annotated formulae in terms of the difference between pairs of formulae. Using this representation of annotated terms, the proof search guidance technique of *rippling* can be extended to higher-order theorems. We illustrate this with selected examples using our implementation of these ideas in λ*Clam*.

Keywords: Higher Order Logic, Automated Theorem Proving, Proof Planning, Rippling

Research funded by EPSRC grant Gr/m45030. We thank members of the Mathematical Reasoning Group in Edinburgh and especially Alan Bundy for input to this work. Some earlier ideas related to this work are due to Jason Gallagher.

1. Introduction

There is an increasing need to automate theorem proving in higher-order (HO) logics. Theories for hardware and software synthesis and verification are typically expressed in HO logics. The encoding of propositional and first-order (FO) logics in logic frameworks means that proof-search is lifted away from the object logic into a HO setting. In this paper we address the problem of controlling proof search in such a setting; in particular our approach is a development of techniques currently used to control first-order term rewriting.

Here the heuristic of *difference reduction* is central: find the differences in term structure between the goal G and the assumptions H_n; then rewrite G in an effort to remove those differences in term structure, in order that an appeal to one or more of the assumptions can be made. Differences are called *wave-fronts* and shared term structure is called the *skeleton*.

Rippling is difference reduction by the application of certain, well-constrained rewrite rules. The key idea of rippling is that terms are decorated with annotations that describe explicitly wave-fronts and skeletons; these annotations can be used to place heuristic restrictions on rewriting. Thus the way in which the differences are manipulated by the rewrite rules can be controlled. By insisting that applicable rewrite rules are *skeleton-preserving* rippling is further restricted. Skeleton preservation ensures that if all the differences are removed, the assumption(s) can be used.

Basin and Walsh (Basin and Walsh, 1996) and Hutter (Hutter, 1991) have developed first-order rippling calculi: formal systems for manipulating annotated terms. However, there are some difficulties in extending such calculi to the λ-calculus, as is required when, for example:

- rippling first-order terms containing meta-variables that stand for functions, and, more generally,

- rippling in logics containing the λ-calculus.

The crux of the problem is that annotations depend on consistent naming of bound variables between terms which causes obvious problems with λ-abstractions. Annotations (first-order as well as higher-order) also require non-standard notions of substitution and unification. Our solution is a new formalisation of rippling in which annotated terms are captured via *term embeddings* which allow the standard definitions of substitution and unification to be used. We show how term embeddings can be defined uniformly for both first- and higher-order syntax, then how term embeddings capture rippling. We believe this presentation provides a clear account of rippling that generalises to the higher-order case.

1.1. Overview.

We discuss embeddings in an informal fashion in a first order setting in §2 and then in a higher-order setting in §3. We then provide some formal definitions in section §4 and discuss the rippling heuristic in §5. We close by looking at some examples of the rippling heuristic using embeddings in §6 and then discuss related work (§7) and our conclusions.

2. Term Embeddings: the first-order case

The notion of an *embedding* (or homeomorphic embedding) between trees is known from its use within rewriting (see (Klop, 1992, p. 31)). For our purposes, following (Boudet and Comon, 1993), we use a slightly different notion, where the order of the subtrees at a node is taken to be significant, and where the labels of a node and its image under an embedding are required to coincide.

More precisely, an embedding between terms t_1, t_2 is defined as follows. Consider terms t_1, t_2 in a standard first-order syntax as labelled trees in the usual way. Then an embedding is an injective map e from the nodes of t_1 to the nodes of t_2 which maps labels to identical labels, which preserves the tree order, and which also preserves "horizontal order". Such an embedding, when it exists, can be represented by a tree of the shape of t_1 labelled with the addresses of the corresponding image nodes in t_2; where e is a tree representing an embedding, we write $e : t_1 \hookrightarrow t_2$, or simply $t_1 \hookrightarrow t_2$ if the identity of e is not important. In what follows the term on the left of an embedding, t_1 above, will be referred to as the *skeleton* of the embedding and the term on the right, t_2 above, as the *erasure.*

This notion can be captured inductively.

DEFINITION 1. *For terms* $t_1 \equiv f(u_1, \ldots, u_n)$ *and* $t_2 \equiv g(v_1, \ldots, v_m)$,

$$\begin{aligned} t_1 \hookrightarrow t_2 \ \textit{ iff } \ & \exists i.\ \ t_1 \hookrightarrow v_i \\ & \textit{or } f \equiv g \textit{ and } \forall i.\ \ u_i \hookrightarrow v_i. \end{aligned}$$

This definition forms the basis for the computation of embeddings, by returning an appropriately labelled tree; this tree has the same shape as the term t_1, and the labels give the address of the image of the node under the embedding map. For an example, consider the terms $plus(x, y)$ and $plus(s(x), y)$ shown in figure 1. In this figure solid lines indicate the term trees, the dashed line the embedding tree and the dotted lines show how the embedding relates the nodes of the two term trees.

The pair of terms together with the embedding describe an *annotated term*, namely t_2 decorated so as to indicate which parts of the term tree are in the image of the embedding, and which are not; in this way we are able to capture the "differences" between terms (more on this in §5).

In earlier descriptions of rippling (e.g., (Bundy et al., 1993)), annotated terms are depicted by placing boxes with holes in them around differences. The holes are underlined. The boxes are referred to as *wave fronts* and the holes as *wave holes.* In this presentation the skeleton

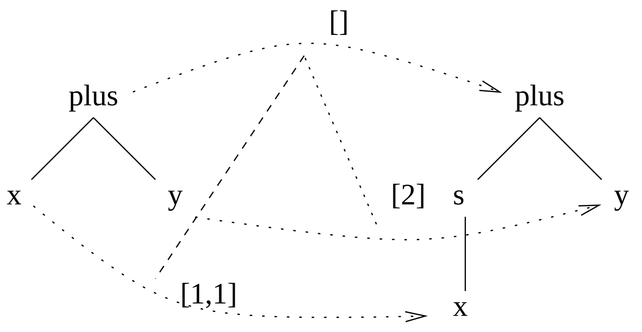

Figure 1. An embedding

is formed by those parts of the term in wave holes or outside of the wave fronts. For example, the embedding shown in figure 1 would be $plus(\boxed{s(\underline{x})}\,, y)$. We shall adopt this notation for readability where is it convenient to do so.

In this earlier work, operations on annotated terms were defined on an enlarged term algebra, with extra functors wrapped around subterms to indicate which parts of a term belong to a skeleton. For example, the term above corresponds to $plus(wf(s(wh(x))), y)$. This notation necessitates the use of non-standard matching and substitution, for cases where the annotational syntax has to be distinguished from the underlying term structure.

2.1. Substitution and Unification

When reasoning with annotated terms, we will use the annotations for search guidance; the annotations should not however interfere with the standard logical operations on the underlying unannotated term. Thus we want to define substitution on embeddings so that it is sound with respect to the substitution on standard terms.

DEFINITION 2. *We define two functions erase and skel (for the erasure and skeleton) from annotated terms to plain terms by*

$$erase(e : t_1 \hookrightarrow t_2) =_{def} t_2,$$

and

$$skel(e : t_1 \hookrightarrow t_2) =_{def} t_1.$$

Now suppose that σ is a substitution on terms.

DEFINITION 3. *We say that an embedding e'* extends *e if the tree e' is obtained from e by replacing some of the leaves of e with new trees, while preserving all the labels within the original tree e.*

It is easy to show the following

PROPOSITION 1. *If $e : t_1 \hookrightarrow t_2$ is an annotated term, and σ a substitution, then there is a unique $e' : t_1\sigma \hookrightarrow t_2\sigma$ such that e' extends e.*

Proof sketch follows in §4.

We therefore define

DEFINITION 4.

$$(e : t_1 \hookrightarrow t_2)\sigma =_{def} e' : t_1\sigma \hookrightarrow t_2\sigma,$$

where e' is this unique extension of e. In general, there can be other embeddings of $t_1\sigma$ in $t_2\sigma$.

From this definition, it is easy to see that *skel* and *erase* respect substitution. That is,

PROPOSITION 2. *For all embeddings $e : t_1 \hookrightarrow t_2$ and substitutions σ,*

$$\begin{aligned} erase((e : t_1 \hookrightarrow t_2)\sigma) &= (erase(e : t_1 \hookrightarrow t_2))\sigma, \\ skel((e : t_1 \hookrightarrow t_2)\sigma) &= (skel(e : t_1 \hookrightarrow t_2))\sigma. \end{aligned}$$

DEFINITION 5. *A* unifier *for two annotated terms, $e : t_1 \hookrightarrow t_2$, $f : u_1 \hookrightarrow u_2$, is thus in the usual way a substitution σ such that*

$$(e : t_1 \hookrightarrow t_2)\sigma = (f : u_1 \hookrightarrow u_2)\sigma,$$

where the equality entails equality of domain terms, and of range terms.

From our previous remark, a unifier in this sense is clearly also a unifier for the erasures of the embeddings, i.e. for t_2, u_2 and for the skeletons t_1, u_1.

DEFINITION 6. *A most general unifier for $e : t_1 \hookrightarrow t_2$, $f : u_1 \hookrightarrow u_2$, when it exists, is simply a most general unifier for the unification problem*

$$\langle t_1 = u_1, t_2 = u_2 \rangle,$$

provided that the corresponding embeddings coincide.

Thus annotated unification is a *restriction* of standard unification, which can be captured for example by a generate-and-test procedure.

3. Embeddings: a higher-order version

When reasoning about languages with variable-binding constructs, it has become normal to deal with binding and substitution once and for all in such a way that the operations can be applied uniformly over a range of situations; see for example (Harper et al., 1992; Felty, 1993).

One challenge is to handle variable bindings in a consistent fashion. Earlier attempts at annotations of differences tended to rely on a consistent naming of bound variables in the skeleton and the erasure in order to achieve annotation, i.e. they could not apply difference matching (Basin and Walsh, 1992) – the process by which a term is annotated – to $\forall x.\ f(x)$ and $\forall y.\ f(g(y))$. It was also unclear how terms such as $\lambda x.\ f(x)\,a$ and $\lambda x.\ f(g(x))\,a$ should be difference matched: The standard algorithms would not find a match since the function terms $\lambda x.\ f(x)$ and $\lambda x.\ f(g(x))$ were not identical. The use of embeddings as presented here can be extended to handle bound variables in a natural fashion and it easily extends to a notion by which for an application term $t_1(t_2)$ it is acceptable to have an embedding into t_1 as well as t_2 instead of insisting that the embedding is only into t_2.

Informally we allow $t \hookrightarrow t$ for atomic t, $t \hookrightarrow \lambda x.u$ iff $\forall z.\ t \hookrightarrow (\lambda x.\ u)z$ or $t = \lambda y.\ t_1$ and $\forall z.\ (\lambda y.\ t_1)z \hookrightarrow (\lambda x.\ u)z$. Lastly $t \hookrightarrow u_1(u_2)$ iff $t \hookrightarrow u_1$ or $t \hookrightarrow u_2$ or $t = t_1(t_2)$ and $t_i \hookrightarrow u_i$.

This approach apparently raises some new problems. For instance we would not expect an operation such as β-reduction to affect the existence of an embedding, as we do not want to distinguish between inter-convertible terms. Since there is no logical significance to the difference between such terms, it would seem unnatural for a proof guidance technique to distinguish between them. In particular this means that the skeleton of the embedding expression is not preserved by substitution and skeleton preservation is a critical component of many applications of annotations.

For example, taking a and b to be constant terms, a embeds in $(\lambda x.\ b)\ a$ according to our initial definition, yet a is not embedded in the reduct b.

This problem also afflicts the other major alternative higher-order annotation calculus for Rippling: the coloured λ-calculus of Hutter and Kohlhase (Hutter and Kohlhase, 2000) which we discuss in §7.

Intuitively we would expect annotation systems to be insensitive to β-reduction. However both embeddings and the coloured λ-calculus annotate the syntactic structure of the term and since β-reduction is a structure effecting process it seems almost inevitable that in some cases it will not preserve the embedding. As a result of this we also discover that skeletons are not preserved by substitution. Ideally if

$t_1 \hookrightarrow t_2$ then for all substitutions σ we want $(t_1)\sigma \hookrightarrow (t_2)\sigma$. The following example shows that we lose this property if β-reduction does not preserve embeddings:

EXAMPLE 1. *There are terms t_1, t_2, and substitution σ such that there is an embedding $e : t_1 \hookrightarrow t_2$, yet we do* ***not*** *have $t_1\sigma \hookrightarrow t_2\sigma$.*

If we take $t_1 = p(q)$, $t_2 = (C(p))(q)$, where p, q are constants, and C a variable, then there is an embedding $e_1 : t_1 \hookrightarrow t_2$. However, if we substitute $\lambda x.\ \lambda y.\ (D(y))(x)$ for C, we find after normalisation that there is no embedding of $t_1\sigma$ in $t_2\sigma$. A consistent typing can be provided for the terms used in this example, e.g. $q : nat$, $p : nat \to nat$, $C : (nat \to nat) \to (nat \to nat)$, $D : nat \to (nat \to nat) \to nat$

There are a number of possible solutions to this problem. For instance, Hutter and Kohlhase consider including all function terms in the skeleton and restricting admissible solutions. In the end they settle for a test of skeleton preservation after a substitution has been made in circumstances where this is important. We chose to separate the processes of substitution and β-reduction. If no normalisation can occur during substitution then the proof that the process of substitution preserves embeddings follows straightforwardly from the first order case (see §4).

Much like Hutter and Kohlhase we then check for the preservation of embeddings whenever we apply β-reduction. All the applications of the embeddings calculus so far have applied to rewriting processes. For pragmatic reasons we found we wanted to treat β-reduction as a rewrite rule and we already had an embedding preservation check whenever a rewrite rule was applied. So separating the processes worked best within our implementation. Hutter and Kohlhase annotate their rewrite rules as well as the terms on which they operate and rely on this annotation to preserve the skeleton in their situation therefore a check for preservation at the substitution stage makes more sense.

4. Formalising Embeddings

In first-order presentations of annotations (e.g. (Basin and Walsh, 1996)) a term tree representation is used in which nodes in the tree are labelled with function symbols and the leaves are all labelled with constants or variables of first-order type. We do not do this since we potentially wish to be able to manipulate function symbols in the same way the first order variations can manipulate first-order symbols.

We treat the syntax of terms as a variant of Higher-Order Abstract Syntax or λ-tree syntax (Miller, 2000) in which we explicitly represent abstraction and application as nodes in the term tree. We use this for a variety of reasons, principle among these is the fact that function-argument and term-subterm relationships are made explicit. In particular our term trees needed to be able to represent structure within both constituents of an application, not just the argument to the function. This becomes important when, for instance, calculating the ripple measure on embeddings since wave annotations may appear within function terms. This differs from our original presentation of higher order embeddings in (Smaill and Green, 1996) where we used a concrete syntax however an abstract syntax is more natural in the context of embedding trees since it directly represents the structure used by the embeddings. Our description and results can be relatively easily transferred between concrete and abstract syntax. In general we will use concrete syntax in this presentation since it is easier for a human to read and understand.

There is an example of this approach in (Felty, 1992) although we allow tuples in application terms. The terms of interest are thus the typable terms of this calculus, formed from the constants via typed λ-abstraction, composition, and tupling.

Our approach in this paper differs from Felty's in one important respect. Felty represents the λ-abstraction constructor as of type `((term -> term) -> term)` this causes well documented problems with exotic terms (see (Despeyroux and Leleu, 1999) for an example of the problem and (Felty, 2002) for Felty's approach to a solution). For this reason we treat it here as it is used in the PROSPER PII (Dennis et al., 2003) where the constructor takes a pair consisting of the bound variable and a term[1].

We use an explicit tree representation for our terms which we've defined as follows:

DEFINITION 7. *A* term tree*, tt, is:*

$$tt ::= c \mid X \mid \lambda_x tt \mid app(tt, [tt, \ldots, tt]).^{2}$$

The use of tuples within application terms here is for efficiency reasons. It is normal to use curried syntax in a higher-order system, and our definition allows this; however this means that a given term has typically many more subterms than in a standard first-order representation, and it is often convenient to ignore such terms by representing tuples via products.

[1] Our actual implementation follows Felty closely and relies on the execution model of λProlog to prevent the use of unevaluated functions within the term structure.

[2] We use the Prolog convention where lower case letters indicate constants and upper case letters variables.

The position of a node in a term tree is determined by a list of integers indicating the path from the root of the tree to that node (for implementation reasons this list is usually read from right to left). We label the operator of an application term as the 1st branch and then each member of the ensuing tuple from 2 to n in order and ignore λ-abstraction nodes. So the position of g in the term $g(\lambda x.f(a, x, b))$ is [1], the position of x is [3,2] and the position of f is [1,2].

4.1. Motivation for the Embedding Structure

Embeddings are described by a tree data structure in an extension of the first order case. Embedding trees describe how the skeleton term tree embeds in the erasure term tree. The nodes in an embedding tree can be viewed as labels on the nodes in the term tree of the skeleton (excluding λ-abstraction nodes). These labels contain addresses. The addresses are the addresses of nodes in the term tree of the erasure into which the skeleton is to be embedded. 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 erasure term tree at the indicated address. Similarly the leaves of an embedding tree are attached to the leaves of the skeleton term tree.

EXAMPLE 2. *Consider embedding the term $\lambda x.\ f(x)$ into the term $\lambda y.\ \lambda x.\ (f(y)+x)$. 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 (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.*

4.2. Definition of Embeddings

DEFINITION 8. *An embedding tree, e, is*

$$e ::= \texttt{leaf}(pos) \mid fn(pos, e, [e, \cdots, e]). \tag{1}$$

Note that each node in the embedding is annotated with the position of the node in the erasure to which that node in the embedding relates. It is not necessary to include the position information for the skeleton since the embedding tree has the same structure as the term tree of the skeleton. These positions are lists of integers read from right to left as described above.

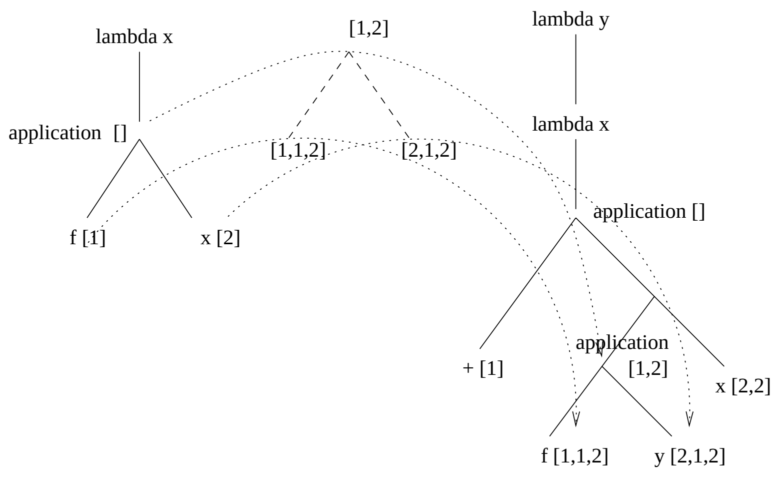

Figure 2. An Embedding

We define a function *epos* for extracting the address information from embeddings in the obvious way (i.e. $epos(\texttt{leaf}(pos)) = pos$, $epos(fn(pos, e, \bar{e})) = pos$).

NOTATION: We use the notation $\bar{t^n}$ to represent the expression $[t_2, \cdots, t_n]$.

DEFINITION 9. *$e : s \hookrightarrow (t, pos)$ where e is an embedding, s and t are term trees and pos is a term tree node position is an* embedding expression*.*

An embedding expression is a well-embedding *(*we*) iff the statement $we(e)$ can be derived in the following inference system:*

$$\frac{}{we(\texttt{leaf}(pos) : c \hookrightarrow (c, pos))}, \tag{2}$$

$$\frac{}{we(\texttt{leaf}(pos) : X \hookrightarrow (X, pos))}, \tag{3}$$

$$\frac{\forall y.\ we(e : f(y) \hookrightarrow (g(y), epos(e)))}{we(e : \lambda x. f(x) \hookrightarrow (\lambda x.\ g(x), epos(e)))}, \tag{4}$$

$$\frac{pos \neq epos(e) \qquad \forall y.\ we(e : s \hookrightarrow (f(y), pos))}{we(e : s \hookrightarrow (\lambda x.\ f(x), pos))}, \tag{5}$$

$$\frac{\forall i.\ we(e_i : s_i \hookrightarrow (t_i, i :: pos))}{we(fn(pos, e_1, \bar{e^n}) : s_1(\bar{s^n}) \hookrightarrow (t_1(\bar{t^n}), pos)}, \tag{6}$$

$$\frac{epos(e) \neq pos \qquad \exists i.\ we(e : s \hookrightarrow (t_i, i :: pos))}{we(e : s \hookrightarrow (t_1(\bar{t^n}), pos))}, \tag{7}$$

Obvious rules are used to handle quantification over well-embeddings.

It follows from this definition that in a well-embedding every λ-abstraction in the skeleton is matched by a corresponding λ-abstraction in the erasure. These definitions correspond to the informal outline given in §3. Note that we have extended our embedding notation, $\hookrightarrow$, to include the position information. This assumes that the erasure is a subterm of some larger term and this indicates its position within that larger term. In most cases we will be considering embeddings at the root node (i.e. $e : s \hookrightarrow (t, [])$) but during proofs and implementations we frequently need to recurse through the structure of terms and this additional information becomes important. Where the position information is omitted it can be assumed to be the root node.

The abstraction case here reduces the problem at functional types to that of finding embeddings for atomic types, at the cost of requiring that they exist uniformly. This is the method that is used to deal with the case of standard variable binding constructs such as quantifiers.

The embedding tree for example 2 is thus $fn([1,2], \texttt{leaf}([1,1,2]), [\texttt{leaf}([2,1,2])])$. 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 [1,1,2] (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).

EXAMPLE 3.
In §6.1 we will see an example of rippling involving an introduced constant lim*, and some additional symbols:*

$$\begin{aligned}
\lim &: (nat \to nat) \to nat \to nat \to nat,\\
f &: nat \to nat,\\
g &: nat \to nat,\\
+ &: (nat \times nat) \to nat,\\
a &: nat,\\
l &: nat,\\
m &: nat.
\end{aligned}$$

An embedding exists between the terms $lim(f, a, l)$ and $lim(\lambda u.\ f(u) + g(u), a, l + m)$. In our higher order abstract syntax this appears as:

$app(\lim,\ [f,\ a,\ l]) \hookrightarrow$
$app(\lim,\ [(\lambda_u(app(+, [app(f,\ [u]),\ app(g\ [u])]),\ a,\ app(+, [l,\ m])]),$

The embedding exists by virtue of the fact that for all terms z,

$$(\lambda y.\, f(y))\, z \hookrightarrow (\lambda u.\, f(u) + g(u))\, z.$$

The embedding in question is

$$fn([], \texttt{leaf}([1]), [\texttt{leaf}([1,2,2]), \texttt{leaf}([3]), \texttt{leaf}([2,4])])$$

4.3. Subtitution

Recall that from §2 we need to show how substition acts on embedding expressions.

DEFINITION 10. *We say that an embedding e'* extends *e if the tree e' is obtained from e by replacing some of the leaves of e with new trees, while preserving all the labels within the original tree e.*

Recall that we are not allowing normalisation to take place in the course of substitution so in t_1 and t_2 we are simply extending leaf nodes representing variables with further term trees. No other alteration of the structure of the term trees can take place. In fact we are using the definition of substitution from (Barendregt, 1985, §2.1.15) directly. It has simply become customary to implement a β-reduction as a part of substitution from a desire to work with terms in β normal form as much as possible.

Armed with our formal definitions of embedding our Proposition 1 becomes:

PROPOSITION 3. *If $e : t_1 \hookrightarrow (t_2, pos)$ is a well-embedding, and σ a substitution, then there is a unique well-embedding $e' : t_1\sigma \hookrightarrow (t_2\sigma, pos)$ such that e' extends e.*

Proof Sketch. It is easy to define a unique *identity embedding*, e^I_t for a term t such that $e^I_t : t \hookrightarrow (t, [])$ is a well-embedding. Let $c_p(e, pos)$ be a function which appends the integer list pos to the start of all the position information in e it follows that $c_p(e^I_t, pos) : t \hookrightarrow (t, pos)$ is also a well-embedding.

Recall that we are not allowing normalisation to take place in the course of substitution so in t_1 and t_2 we are simply extending leaf nodes representing variables with further term trees. No other alteration of the structure of the term trees can take place.

It is simple to show that if a leaf node of the embedding is a well-embedding $\texttt{leaf}(pos) : X \hookrightarrow (X, pos)$ where σ replaces X with some term t then $c_p(e^I_t, pos) : t\sigma \hookrightarrow (t\sigma, pos)$ is the unique well-embedding that extends $\texttt{leaf}(pos)$.

The rest of the proof is then a very straightforward inductive argument on the structure of well-embeddings. □

Definitions for unification follow in the obvious way from those given for the first-order case.

5. Rippling: a proof search technique

We now illustrate the use of terms annotated by means of embeddings to guide proof search. This approach to the search for proofs involving induction was proposed in (Bundy, 1988) and has since been developed and also applied to non-inductive proofs, where a logical connection is sought between two syntactically similar formulas (Bundy et al., 1993; Yoshida et al., 1994; Hutter, 1996).

Rippling is a terminating heuristic. It is based on the idea of difference reduction and its termination depends upon a measure on annotated terms. Rippling may either be *static* using specially annotated *wave rules* as rewrite rules and requiring that the annotations in the wave rules unify with those in the term or it may be *dynamic* where rewrites are not annotated in advance. In static rippling measure reduction is calculated in advance for each wave rule while in dynamic rippling the measure is explicitly used as part of the rippling process. The measure expresses the heuristics used to control rippling – namely to move differences *outward* in the term structure (i.e. closer to the root node of the term tree) in the hopes of exploiting cancellation in some fashion or, failing that to move them *inward* in the hope of exploiting universally quantified variables. Rippling in a first order setting has been formalised by (Basin and Walsh, 1996) and the termination of the measure proved. It is therefore important to show how any measure we provide for embeddings corresponds in the first order to case to that of Basin & Walsh in order to demonstrate the relationship between the existing rippling heuristic and that created by embeddings.

5.1. The Embedding Measure

We introduce a well-founded *measure* on embeddings. We will start by defining the *outward* measure (where outward is a direction that can appear in embeddings corresponding to a difference moving outward in the term structure) on a node in an embedding – the *inward* measure is defined analogously.

Informally the measure of a node in an embedding is the difference between it's *expected depth* (which is the depth of the point in the erasure where the node's parent embeds plus one) – where we would expect the node to embed were there no difference between the skeleton and the erasure at this point – and the depth of the point where the node actually embeds. The depth of the point in the erasure where a node embeds is the length of the position list $epos(e)$. It is generally easier to compute the node measures for a whole embedding tree at once, storing the positions at which the parent nodes embed and passing these as arguments to child nodes.

Our measure function therefore has three arguments, the embedding, the expected depth and the address of the node for which we wish to calculate the measure.

DEFINITION 11. *The* weight *function, $\mathcal{W}_e(e, ed, p)$, of a node at position p in an embedding e where its expected depth is ed is defined by:*

$$\mathcal{W}_e(e, ed, []) = length(epos(e)) - ed, \tag{8}$$
$$\mathcal{W}_e(fn(pos, e_1, \bar{e}^n), ed, i :: pos_s) = \mathcal{W}_e(e_i, length(pos) + 1, pos_s). \tag{9}$$

This gives us the difference between the expected address of the erasure node of a sub-embedding (assuming there are no differences at that node) and the actual address of the erasure node.

NB. This is not the same weight function as that published in (Smaill and Green, 1996) which did not, in the first order case, correspond to the weights used by (Basin and Walsh, 1996).

To calculate the weight of a whole embedding rather than a single node we sum the weights of the nodes at each depth in the term tree and present these as a list. The weight lists for two embeddings can then be compared in reverse lexicographic order – so terms are considered measure reducing if more differences appear closer to the root node.

To accommodate the rippling heuristic we incorporate an indication of *inward* or *outward* direction of a difference. We use the notation $\boxed{s(\underline{x})}^{\downarrow}$, $\boxed{s(\underline{x})}^{\uparrow}$ respectively. It is easy to extend embedding nodes with additional direction information and to adapt the weight function to an outward and an inward weight function $\mathcal{WO}_e$ and $\mathcal{WI}_e$. These return 0 if the direction of the embedding node is inward (in the case of $\mathcal{WO}_e$) or outward (in the case of $\mathcal{WI}_e$). Otherwise they are identical to the simple function shown above.

In harmony with this, we define an extended well-founded measure, which is a lexicographic combination of a reverse lexicographic ordering on measure lists (for outbound differences) and a lexicographic ordering on measure lists (for inbound differences).

DEFINITION 12. *The* embedding out-measure, *$\mathcal{MO}_e(e : s \hookrightarrow t)$ of an embedding term is a list of length $|t| + 1$ whose i-th element is the sum* *of out-weights ($\mathcal{WO}_e(e, 0, p)$) for all skeleton positions, p in $e : s \hookrightarrow t$ at depth i (i.e. the position lists p of length i). The* embedding in-measure, *$\mathcal{MI}(t)$ is a list whose i-th element is the sum of in-weights for all term positions in $e : s \hookrightarrow t$ at depth i. The* measure *of an annotated term, $\mathcal{M}_e(e : s \hookrightarrow t)$ is the pair of out and in-measures, $< \mathcal{MO}_e(e : s \hookrightarrow t), \mathcal{MI}_e(e : s \hookrightarrow t) >$.*

NOTATION: For two embeddings, e_1 and e_2 if e_1 is less according to the measure than e_2 we write $e_1 <_{\mathcal{M}} e_2$.

It is a relatively simple exercise to provide functions to convert first-order terms in our syntax into the concrete syntax employed by Basin & Walsh (Basin and Walsh, 1996). This can be seen in appendix A. We can show that, for first order terms t_1 and t_2 represented as uncurried term trees (i.e. with full use of tuples within application nodes), if $t_1 \hookrightarrow t_2$ then it is possible to create an *annotated term*, in the terminology of Basin & Walsh, whose skeleton is t_1 and whose erasure is t_2. Moreover given two well-embeddings $e_1 : t_1 \hookrightarrow u_1$ and $e_2 : t_2 \hookrightarrow u_2$ (such that t_1, t_2, u_1, u_2 are first order and uncurried) then if $e_2 <_{\mathcal{M}} e_1$ then Basin & Walsh's measure on the annotated terms also reduces. In fact the measure is the same. This gives us confidence that our embeddings capture the essence of rippling.

Proof sketch. Basin & Walsh's measure represents the skeleton of an annotated term as a tree with the much the same structure as our skeleton term trees assuming that our representation is uncurried and contains no λ-abstractions. Their nodes are labelled with the functions applying at that node while we use explicit application nodes – however this is irrelevant in a first order setting since there are no wave fronts in the structure of the functions and these might as well annotate the application nodes. Basin & Walsh also annotate the nodes where relevant to indicate the wave fronts at that node in the annotated term. The weight of each node in this tree is then the width of the wave fronts at that node (the number of function symbols contained within the wave front). This width corresponds directly to the difference between the actual depth and expected depth of the embedding of that node in our representation since each function symbol is represented by a function application node in our representation of the erasure and so increases the depth by one. The resulting inward and outward measures are then built up from the weight of the individual nodes in exactly the same way. □

It is interesting to note that the same does not hold the other way around. It is possible to have an annotated term of the form $\boxed{f(\underline{\underline{\boxed{g(\underline{x})}^{\downarrow}}})}^{\uparrow}$ which we cannot express as an embedding with directions. The embedding is $leaf([2,2])$ and this can not be extended to show that there is both an inward and an outward moving wave front.

This means that the following rewrite rules allow a measure reducing series of rewrites from $l(h(s(x)))$ to $l(k(f(h(x))))$ in Basin & Walsh's calculus but not in the embeddings calculus:

$$h(s(X)) \Rightarrow g(f(h(X))) \tag{10}$$
$$l(g(f(X))) \Rightarrow l(f(g(X))) \tag{11}$$
$$f(g(h(X))) \Rightarrow k(f(h(X))) \tag{12}$$

These rules allow the following sequence of ripples:

$$l(h(\boxed{s(\underline{X})}^{\uparrow})) = l(\boxed{(g(f(\underline{h(X)})))}^{\uparrow}) \tag{13}$$

$$l(\boxed{g(f(\underline{h(X)}))}^{\uparrow}) = l(\boxed{f(\boxed{g(\underline{h(X)})}^{\downarrow})}^{\uparrow}) \tag{14}$$

$$l(\boxed{f(\boxed{g(\underline{h(X)})}^{\downarrow})}^{\uparrow}) = l(\boxed{k(f(\underline{h(X)}))}^{\downarrow}) \tag{15}$$

Our calculus would be forced to annotate $l(\boxed{f(\boxed{g(\underline{h(X)})}^{\downarrow})}^{\uparrow})$ as either $l(\boxed{f(g(\underline{h(X)}))}^{\uparrow})$ which is not measure less than $l(\boxed{g(f(\underline{h(X)}))}^{\uparrow})$ or as $l(\boxed{f(g(\underline{h(X)}))}^{\downarrow})$ which is not measure greater than $l(\boxed{k(f(\underline{h(X)}))}^{\downarrow})$. In practice we have not yet come across a situation where a sequence of rewrites such as this is required[3].

5.2. Rippling and Induction

In proofs by induction, using a constructor formulation, the measure is used as follows. Given an induction hypothesis $\Phi(x)$ and goal $\Phi(s(x))$, there is an embedding $e : \Phi(x) \hookrightarrow \Phi(s(x))$. Recursion equations and lemmas are available as rewrite rules to be used on the goal — rewriting here is extended to take account of quantification. Typically, these

[3] NB. There are simpler examples which exhibit similar properties but many of them could be fixed by conjecturing simple additional lemmas from the rewrite rule set alone or forbidding the use of rewrites which were not measure reducing themselves between their LHS and RHS once the annotations on the target term were taken into account even if the term itself could be measure reduced by moving annotations not affected by the rewrite rule. This was the simplest counter-example we could find which did not have an obvious fix.

rewrite rules are not confluent, and include potentially looping rules, such as associativity. The following heuristic can be used: use rewrites which give a new goal G' such that there is an embedding $e' : \Phi(x) \hookrightarrow G'$, and such that $e' <_{\mathcal{M}} e$, *i.e.* the embedding is smaller in the measure defined above.

This gives at each step the picture shown in figure 3 (where e, e' are embeddings and the vertical arrow indicates rewriting). We call such a rewriting step a *wave rewrite*, following the terminology of (Bundy, 1988); the successive application of such wave rewrites is known as *rippling*.

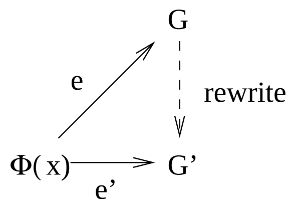

Figure 3. An Embedding for Induction

There are three conditions on wave rewriting:

1. *soundness* – the rewriting should correspond to logically valid inference;

2. *skeleton invariance* — the appropriate skeleton embeds in the goal, before and after rewriting;

3. *measure decreasing* — the embedding in the rewritten formula is smaller in the order.

Traditionally wave rewriting has been accomplished using special annotated *wave rules* in which one rewrite becomes several wave rules each indicating how a different skeleton is preserved by the rule and how the annotations are affected by its application, however since an unannotated erasure is easily extracted from our embedding expressions we choose to use standard rewrite rules and subsequently check after application that there is still an embedding between the skeleton and the new erasure and that the measure of this embedding has decreased. We generate rewrite rules both from equalities and (less usually) from implications.

Since we are reasoning backwards, there is a polarity inversion to the direction in which rules based on implication may be used. A logical implication $A \rightarrow B$ may be used as a rewrite rule $A \Rightarrow B$ (at positions of negative polarity) and $B \Rightarrow A$ (positive polarity). Equalities are used in either direction. In all cases the direction of the wave-rule is given by the double arrow $\Rightarrow$[4]. Our implementation of polarity is based on work by Negrete and Smaill (Negrete and Smaill, 1995).

This approach has considerably simplified the theory of rippling – for instance there is no need to show that if a wave rule is skeleton preserving (i.e. the LHS and RHS of the equation share the same skeleton) then the skeleton of any term rewritten with the rule is also preserved. It also potentially allows us to exploit standard rewriting technology alongside the use of rippling as an extra guidance procedure and may allow the rapid transfer of results about rewriting to rippling. It also helps to make clear the relationship between rippling and rewriting.

We can depict the possible ways a wave rewrite may be made on the goal by considering the rewrite rules available, and annotating them with embeddings: these annotated rules are called *wave-rules.* In general there is more than one way in which this annotation can be added, and it may be added so that the rule can be used in either direction.

EXAMPLE 4. *We will see in §6 some example proofs using rippling. There use is made of the associativity of disjunction:*

$$A \vee (B \vee C) \equiv (A \vee B) \vee C,$$

which can be used as a rewrite in either direction, depending upon the skeleton to be preserved, and the direction in which the wave-front is to be moved. For example, the two wave-rules below are derived from the associativity property:

$$A \vee \boxed{(\underline{B} \vee C)}^{\uparrow} \;\Rightarrow\; \boxed{\underline{(A \vee B)} \vee C}^{\uparrow},$$
$$\boxed{(A \vee \underline{B})}^{\uparrow} \vee C \;\Rightarrow\; \boxed{A \vee (\underline{B \vee C})}^{\uparrow}.$$

In the first rule the skeleton is $A \vee B$ (which embeds into the LHS and RHS of the wave-rule), and in the second it is $B \vee C$ (also embedded in both sides). Thus one is able to use rewrites in both *directions during rippling so long as they are in contexts where the skeletons differ.*

Rewriting in a higher order setting can be done using the ideas of (Felty, 1992). After applying a rewrite rule to an erasure, we recalculate the embedding with respect to the skeleton and check for measure reduction, in experiments we have found that this recalculation of embeddings and the measure after a rewrite is not a significant

[4] Note that we are using $\rightarrow$ to indicate logical implication and $\Rightarrow$ to express a rewrite rule.

problem for efficiency although clearly providing annotated, skeleton-preserving and measure reducing wave rules "up-front" would save time during proof[5].

5.3. Sinks

We mention above that the intention when rippling inwards is to exploit a universally quantified variable in the skeleton. We call such variables *sinks* and annotate them as $\forall x.\ f(\lfloor x \rfloor)$. In rippling it is an additional condition on inwards moving wave fronts that there is a potential sink they can use. This extends the set of terms into which a potential skeleton can embed. It does not effect the formalism in terms of the measure but acts as an additional heuristic restriction on rippling. We choose to ignore it in this presentation since it is an addition to the basic theory. Our implementation adds an additional leaf type to embedding trees to represent leaves which are sinks but during rippling treats these just like leaf nodes in all cases except when checking for the sinkability of a wave front.

5.4. Middle Out Reasoning

The use of meta-variables in implemented systems to delay commitment to certain choices in proof search is commonplace (this is used in Lego and Isabelle, for example (Luo and Pollack, 1992; Paulson, 1994)). This is a form of *middle out reasoning.* We use such meta-variables in dealing with existentially quantified goals[6]; the interaction between the instantiation of meta-variables and the wave annotations has been a stumbling block in earlier attempts to extend the scope of rippling.

The use of embeddings enables us to deal with this situation. Here we allow the terms and formulas we manipulate to contain meta-variables. An annotated term is simply an embedding as before, in which either the erasure, or both the skeleton and erasure, contain meta-variables. Now instantiation of such meta-variables is defined for formulas in the standard way for the simply-typed λ-calculus; our results about substitutions and unification assure us that embeddings will be instantiated in a sensible way.

The treatment of termination has to be different in the case where we deal with meta-variables. The proofs of the termination of rippling given for earlier versions (Bundy et al., 1993; Basin and Walsh, 1996)

[5] It is also possible to improve the efficiency of embedding calculation during rewriting by preserving those parts of the embedding tree not affected by the rewriting and only recalculating the embedding on the rewritten subterm.

[6] NB. Unification of such variables is naturally higher-order.

are necessarily restricted to the ground case: some additional mechanism is needed to deal with the middle-out case. The problem is that if one of the rewriting steps causes the instantiation of a meta-variable, as we want, then the resultant embedding will not be smaller in our given measure since new structure will have been introduced into the term trees. The heuristic in this case must provide some way of controlling the instantiation by skeleton preservation; recomputation of the embedding can be used as part of this process.[7]

We note here that we want annotations to *guide* the instantiation through the application of rewrites; we do not allow the computation of embeddings to cause any instantiation on its own. This means that we use the weaker notion of embeddability rather than embedding.

5.5. Coloured Rippling

In (Hutter, 1990; Yoshida et al., 1994) notions of "colouring" annotated terms are used for situations in which a number of different skeletons are in play and so differences are manipulated with respect to more than one term. Colouring can be used to indicate to which skeleton a wave hole relates, or whether it relates to both. We also wish to be able to manipulate differences with respect to more than one skeleton at once. This can be done by simply extending our terminology so that one erasure is related to several skeleton/embedding pairs. At present we guide rippling in this situation by insisting that a rewrite be skeleton preserving and measure reducing with respect to at least one of the pairs and discard any which fail these conditions as rippling progresses.

It would be possible to strengthen this heuristic to insist that all pairs be skeleton preserving and measure reducing but at present we consider this to be too restrictive since the automatic generation of embeddings can find unexpected pairs. For instance, if induction has been invoked several times in the course of the attempt to find the proof then a number of induction hypotheses have been created many of which may embed into a given conclusion but only one of which will actually be used during fertilisation (exploitation of the induction hypothesis).

[7] The heuristic used in the implementation is very simple: an upper bound is placed on the number of ripple steps which *increase* the measure. (See (Smaill and Green, 1995) for a discussion of this heuristic.)

6. Examples, Results

In this section we show some examples to illustrate higher-order embeddings. The examples were carried out using the λ*Clam* proof planning system (Richardson et al., 1998). λ*Clam* 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[8]. We used λ*Clam* version 3.1 (Dennis, 2000) for our experiments and the appropriate code files can be found in the current distribution.

λ*Clam* is a proof planning system. Proof planning in λ*Clam* works as follows: A goal is presented to the system and *proof methods* are applied to this goal to create subgoals. The 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 *proof strategy* 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 the induction conclusion at least once and does not attempt any other proof method (thus reducing the search space at this point). It then tries to fertilise (exploit the induction hypothesis) once it is no longer possible to rewrite the conclusion. Atomic methods have a preconditions and effects (postconditions)[9] 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(s) on which planning should be attempted.

λ*Clam*'s proof strategies can be illustrated diagrammatically to show the sequence of steps, loops, repeats and or-choices. Depth-first search is used to navigate the choices in a strategy to reach a solution.

6.1. LIM+ THEOREM

This theorem was proposed by Bledsoe as a benchmark theorem for automated theorem provers (Bledsoe, 1990). LIM+ has two hypotheses, and the difference reduction proof will exploit both of these.

The definition of lim is as follows:

[8] We use the Teyjus implementation of λProlog.

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

$$\begin{aligned}&\lim(f,a,l) \equiv \\ &\quad \forall \epsilon.\ (0 < \epsilon \rightarrow \exists \delta.\ (0 < \delta \wedge \\ &\quad \forall x.\ (x \neq a \wedge |x - a| < \delta \rightarrow |f(x) - l| < \epsilon))),\end{aligned}$$

and the LIM+ theorem is:

$$\lim(f,a,l) \wedge \lim(g,a,m) \rightarrow \lim(\lambda x.\ f(x) + g(x), a, l + m).$$

Here f and g are functions on the reals, a is a point at which we are taking the limits, and l and m are the limiting values of f and g at a.

We shall use the following HO rewrite rules in the proof:

$$\begin{aligned}(U_1 + U_2) - (V_1 + V_2) &\Rightarrow (U_1 - V_1) + (U_2 - V_2), \\ |U + V| < E &\Rightarrow |U| + |V| < E, \\ U + V < W &\Rightarrow U < \mathit{half}(W) \wedge V < \mathit{half}(W), \\ \forall x.P_1\, x \wedge P_2\, x &\Rightarrow \forall x.P_1\, x \wedge \forall x.P_2\, x, \\ 0 < E \rightarrow P \wedge Q &\Rightarrow 0 < \mathit{half}(E) \rightarrow P \wedge 0 < \mathit{half}(E) \rightarrow Q.\end{aligned}$$

and the rather unusual rule:

$$\begin{aligned}&\exists \delta.\ (P\ \delta) \wedge \forall x.\ (C\ x \wedge Q\ x < \delta) \rightarrow (R_1\ x) \wedge (R_2\ x) \Rightarrow \\ &\quad \exists \delta.\ (P\ \delta) \wedge \forall x.\ (C\ x \wedge Q\ x < \delta) \rightarrow R_1\ x \wedge \\ &\quad \exists \delta.\ (P\ \delta) \wedge \forall x.\ (C\ x \wedge Q\ x < \delta) \rightarrow R_2\ x.\end{aligned} \tag{16}$$

We have based our rewrite system on the axiomatisation given by Bledsoe (Bledsoe, 1990) for first-order resolution theorem provers. There are some differences: Bledsoe uses unary minus and commutativity and associativity of $+$, rather than the distributivity lemma that we use here. Most deviant from his presentation however is rule 16 concerning existential quantification. This rule expresses the fact that we can distribute $\exists \delta$ through $\wedge$ providing the context is appropriate (obviously there is no such distributivity rule in general). The primary justification of this rule is $U < X \wedge U < Y \rightarrow U < \min(X, Y)$ (which Bledsoe admits as an axiom). The need to provide such a rewrite rule is clearly a weakness in the automation of the proof and in our opinion it remains a significant challenge problem.

6.2. The Proof Strategy

The proof strategy used to tackle the problem is shown in figure 4. The strategy repeatedly applies one of symbolic evaluation (which is used primarily to simplify away definitions), quantifier elimination, checking whether the goal is a propositional tautology and the annotated rewriting method, which itself is a compound method which first annotates

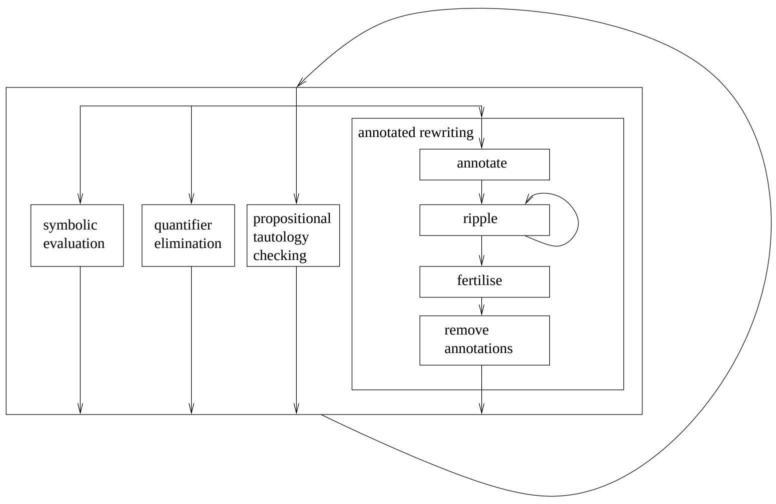

Figure 4. The Proof Strategy for LIM+

the goal with respect to its hypotheses, ripples as much as possible and then fertilises (appeals to the hypotheses).

6.3. The ripple proof.

We start the proof by replacing each occurrence of lim with its definition (using symbolic evaluation), then embedding both conjuncts of the antecedent into the consequent. This results in two embeddings (one for each conjunct). The first of these embeddings annotates the goal as follows:

$$\begin{aligned}&\forall \epsilon.\ (0 < \epsilon \to \exists \delta.\ (0 < \delta \wedge \\ &\qquad \forall x.(x \neq a \wedge |x - a| < \delta \to |\boxed{\underline{f(x)} + g(x)}^{\uparrow} - \boxed{\underline{l} + m}^{\uparrow} | < \epsilon))).\end{aligned}$$

The second is analogous, placing $g(x)$ and m in wave holes.

Exhaustive rippling with the rules above puts the goal into a normal form thus (still showing only the embedding created by the first hypothesis)

$$\begin{aligned}
&\boxed{\forall \epsilon.\ (0 < \boxed{half\,(\underline{\epsilon})}^{\downarrow} \to \\
&\quad \exists \delta.\ (0 < \delta \wedge \forall x.\ (x \neq a \wedge |x - a| < \delta \to |(f\ x) - l| < \boxed{half\,(\underline{\epsilon})}^{\downarrow}))) \wedge} \\
&\underline{\forall \epsilon.\ (0 < half\,(\epsilon) \to} \\
&\quad \boxed{\underline{\exists \delta.\ (0 < \delta \wedge \forall x.\ (x \neq a \wedge |x - a| < \delta \to |(g\ x) - m| < half\,(\epsilon))))}}^{\uparrow}
\end{aligned}$$

at which point the hypothesis may be used. Notice that the presence of the remaining difference (two occurrences of $\boxed{half\,(\underline{\epsilon})}^{\downarrow}$) does not prevent this appeal to the hypothesis because the ϵ in the assumption is universally quantified and so we may use any instance of it. The ability to use universal positions in this way the motivation behind the inwards rippling introduced in §5.

The second conjunct can then be used to fertilise the other conjunct appearing in the conclusion, simplifying the goal to $(T \wedge T)$ (where T is the propositional constant for true) and we are done.

6.4. Program synthesis

This example illustrates the synthesis of functional programs in type theory. Our work is based on a constructive type theory called *Oyster* (Bundy et al., 1990) (which is a close relation to Nuprl (Constable et al., 1986)) although $\lambda Clam$ does not use *Oyster* directly. In this setting the problem of (recursive) program synthesis from specifications is the problem of (inductive) theorem proving: from a proof of

$$\forall input.\ \exists output.\ (spec\ input\ output), \tag{17}$$

a program satisfying the specification *spec* can be extracted.

The strategy used for this is shown in figure 5. This is the standard $\lambda Clam$ strategy for approaching proof by induction. The only difference is that the ripple method is adapted slightly to allow measure increasing steps as discussed in §5. The step case method here is otherwise identical to the annotated rewriting method used in the LIM+ theorem. It is our experience that the method "building blocks" of strategies can often be reused in this way and part of the thrust of our research is to find ever more generic strategies.

On a conjecture of the above form, our synthesis proof strategy is based on that in (Smaill and Green, 1995). It chooses an appropri-

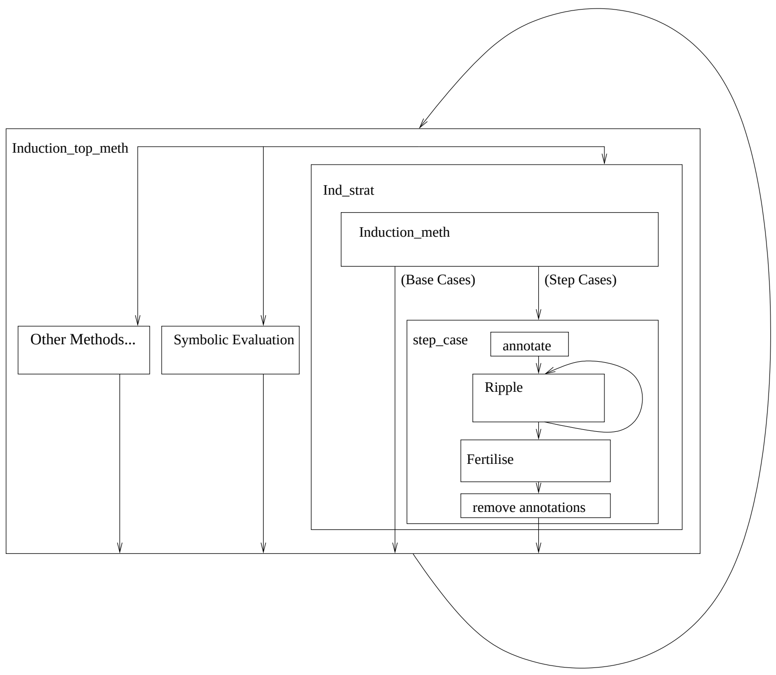

Figure 5. The Induction Strategy

ate induction scheme and then searches for an existential witness using middle-out reasoning. A meta-variable N ranging over object-level terms is introduced to stand for this witness.

$$\forall input.\ (spec\, input\ (N\ input)). \tag{18}$$

As an example we will give a proof of

$$\begin{array}{l}\forall l : list(nat), m : list(nat).\ \exists n : list(nat).\\ \qquad \forall x : nat.\ (x \in l \vee x \in m) \rightarrow x \in n,\end{array}$$

where the following rewrites are available

$$\begin{aligned} X \in nil &\Rightarrow false, & (19)\\ X \in H :: T &\Rightarrow X = H \vee X \in T, & (20)\\ P \vee Q \vee R &\Rightarrow P \vee Q \vee R, & (21)\\ P \vee Q \rightarrow P \vee R &\Rightarrow Q \rightarrow R, & (22)\end{aligned}$$

The proof-plan chooses an $h :: t$ induction on the input list l. This results in the annotated step-case goal

$$\begin{array}{l} m, t : list(nat), h : nat \\ \forall m'.\ \exists n_0.\ \forall x.\ x \in t \vee x \in m' \rightarrow x \in n_0 \\ \vdash \\ \forall x.\ x \in \boxed{h :: \underline{t}}^{\uparrow} \vee x \in m \rightarrow x \in (N\,h\,t\,m). \end{array} \qquad (23)$$

This embedding captures the difference between the induction hypothesis and the goal.

Rippling with (20) then (21) gives

$$\begin{array}{l} m, t : list(nat), h : nat \\ \forall m'.\ \exists n_0.\ \forall x.\ x \in t \vee x \in m' \rightarrow x \in n_0 \\ \vdash \\ \forall x.\ \boxed{x = h \vee \underline{x \in t \vee x \in m}}^{\uparrow} \rightarrow x \in (N\,h\,t\,m). \end{array} \qquad (24)$$

At this point no more ripples are possible: for example, rippling prevents the repeated use of the associativity of $\vee$. Rule (20) is applicable if we allow the measure to increase, as we do according to §5.4. Applying this measure-increasing rule instantiates N by higher-order unification to $\lambda uvw.(N_1\,u\,v\,w) :: (N_2\,u\,v\,w)$. These fresh variables N_1 and N_2 come from the higher-order unification procedure of λProlog: they capture a most general instantiation of N. It is the generality of the substitution which allows subsequent proof to proceed middle-out. (The rule (19) is also applicable at this point, instantiating N to be the constant function returning *nil*. However, this instantiation is rejected since it does not preserve the embedding.)

$$\begin{array}{l} m, t : list(nat), h : nat \\ \forall m'.\ \exists n_0.\ \forall x.\ x \in t \vee x \in m' \rightarrow x \in n_0 \\ \vdash \\ \forall x.\ \boxed{x = h \vee \underline{x \in t \vee x \in m}}^{\uparrow} \\ \qquad\qquad \rightarrow \boxed{x = (N_1\,h\,t\,m) \vee \underline{x \in (N_2\,h\,t\,m)}}^{\uparrow}, \end{array} \qquad (25)$$

N_1 is instantiated to a projection onto its first argument by the application of rule (22). The resulting goal is

$$\begin{array}{l} m, t : list(nat), h : nat \\ \forall m'.\ \exists n_0.\ \forall x.\ x \in t \vee x \in m' \rightarrow x \in n_0 \\ \vdash \\ \forall x.\ x \in t \vee x \in m \rightarrow x \in (N_2\,h\,t\,m) \end{array} \qquad (26)$$

and this can be finished by fertilisation, with the instantiation of N_2 to n_0. Composing the instantiations gives $(N\,h\,t\,m\,n_0) = h :: n_0$. From this proof a functional program for list append can be automatically extracted; different proofs yield different functions still satisfying the specification.

6.5. Other Examples

λ*Clam* contains a benchmark set of over 100 theorems it can successfully plan. The vast majority of these plans involve induction or rippling (or both). This set includes many examples planned using the first-order versions of the rippling heuristic previously proved in the *Clam* system which implemented first-order rippling. Transferring the full *Clam* corpus to λ*Clam* is an ongoing task but so far we have not encountered any problems relating to the new embeddings version of rippling.

A number of case studies have also been conducted exploiting the higher order aspects of λ*Clam* including (Dennis and Smaill, 2001) and (Dennis and Bundy, 2002).

7. Related work

The idea of rippling dates back to Aubin (Aubin, 1975); Alan Bundy is responsible for rippling as we have presented it here, and for the conception of its termination measure and proof-planning in general (Bundy, 1988; Bundy et al., 1993).

An interest in higher-order rippling was initially brought about by the need to treat binding operators correctly, and latterly by the need to ripple essentially higher-order syntax (for example in Bledsoe's LIM family of theorems which were reported in (Yoshida et al., 1994)).

Chuck Liang was the first to integrate rippling into a higher-order language, although the related term rewriting was essentially first-order (Liang, 1992). His representation used λ-abstraction to represent context (the wave-front), the (position of the) wave-hole being marked by the bound variable; the wave-hole term is carried separately. Thus annotated terms have type $(i \to i) \times i$. (There is a wave-operator to map terms of this type into i.) Left- and right-projection deliver the wave-front and hole; applying the former to the latter gives the erasure. This representation suffers from a certain lack of expressiveness and combinatorial complexity when one attempts to generalise it to the coloured case (see (Gallagher, 1993) for details, extensions and generalisations). Liang hints at the difficulty in making sensible HO annotations.

In parallel with the development of rippling in Edinburgh, Dieter Hutter has formulated a sound calculus of annotated FO terms in which one can express rippling (Hutter, 1991). In each of these calculi the underlying term algebra of the logic is extended in order to represent annotated terms. It is our experience that such an approach makes it very difficult to demonstrate the necessary properties of rippling. Furthermore, one is obliged to devise and implement new "extended" calculi on a logic-by-logic basis. Hutter and Kohlhase have extended Hutter's first-order calculus to the λ-calculus (Hutter and Kohlhase, 2000). Hutter and Kohlhase's calculus requires annotated rewrite rules to be provided and so necessitates more complicated theory and in particular non-standard definitions of substitution and unification.

Hutter and Kohlhase point out that the embeddings calculus presented here allows the term (fac) to be embedded into $g(fab)c$ which violates the intended subterm relation. They comment that they foresee that this will cause problems in providing a termination order for rippling which we have not found to be the case and we do not, in principle, feel that embeddings of this form should be forbidden in a full higher-order setting. We know of no examples where an embedding of this kind is either required or should be forbidden and so it is unclear whether or not this should be regarded as a problem or a feature.

Higher-order embeddings treat annotated terms in a more abstract way. Correctness of unification and rewriting are immediate from the given logic. None of the basic logical machinery needs modification.

It would be interesting to see if embeddings and the associated operations can be implemented without calling on full higher-order unification, for example using the restriction to β_0-unification possible with *higher-order patterns* (Miller, 1991; Nipkow, 1991).

8. Conclusion

The use of embeddings to represent annotated terms allows concise and implementable characterisation of wave annotations that is closer to their "intended meaning". It thus allows a systematic approach to the use of annotations to guide proof search in higher-order proof systems and provides a more elegant and extensible description of the process than calculi which rely on an extended term language.

References

Aubin, R.: 1975, 'Some generalization heuristics in proofs by induction'. In: G. Huet and G. Kahn (eds.): *Actes du Colloque Construction: Amélioration et vérification de Programmes.*

Barendregt, H. P.: 1985, *The Lambda Calculus.* Elsevier.

Basin, D. and T. Walsh: 1992, 'Difference Matching'. In: D. Kapur (ed.): *11th Conference on Automated Deduction*, Vol. 607 of *Lecture Notes in Artificial Intelligence.* Saratoga Springs, NY, USA, pp. 295–309.

Basin, D. and T. Walsh: 1996, 'A calculus for and termination of rippling'. *Journal of Automated Reasoning* **16**(1–2), 147–180.

Bledsoe, W. W.: 1990, 'Challenge problems in elementary calculus'. *Journal of Automated Reasoning* **6**(3), 341–359.

Boudet, A. and H. Comon: 1993, 'About the theory of tree embedding'. In: M.-C. Gaudek and J.-P. Jouannaud (eds.): *TAPSOFT '93: Theory and Practice of Software Development.* pp. 376–90.

Bundy, A.: 1988, 'The Use of Explicit Plans to Guide Inductive Proofs'. In: R. Lusk and R. Overbeek (eds.): *9th Conference on Automated Deduction.* pp. 111–120. Longer version available from Edinburgh as DAI Research Paper No. 349.

Bundy, A., A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill: 1993, 'Rippling: A Heuristic for Guiding Inductive Proofs'. *Artificial Intelligence* **62**, 185–253. Also available from Edinburgh as DAI Research Paper No. 567.

Bundy, A., F. van Harmelen, C. Horn, and A. Smaill: 1990, 'The Oyster-Clam system'. In: M. E. Stickel (ed.): *10th International Conference on Automated Deduction.* pp. 647–648. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

Constable, R. L., S. F. Allen, H. M. Bromley, et al.: 1986, *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall.

Dennis, L. A.: 2000, 'User/Programmer Manual for the λ*Clam* Proof Planner v.3.0'.

Dennis, L. A. and A. Bundy: 2002, 'A Comparison of two Proof Critics: Power vs. Robustness'. In: V. A. Carreño, C. A. Muñoz, and S. Tahar (eds.): *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002.* pp. 182–197.

Dennis, L. A., G. Collins, M. Norrish, R. J. Boulton, K. Slind, and T. F. Melham: 2003, 'The PROSPER Toolkit'. *Int. Journal Software Tools for Technology Transfer* **4**, 189–210.

Dennis, L. A. and A. Smaill: 2001, 'Ordinal Arithimetic: A Case Study for Rippling in a Higher Order Domain'. In: R. J. Boulton and P. B. Jackson (eds.): *Theorem Proving in Higher Order Logics: 14th Internional Conference, TPHOLs 2001.* pp. 185–200.

Despeyroux, J. and P. Leleu: 1999, 'Primitive recursion for higher-order abstract syntax with dependant types'. In: *Proc. of FLoC'99 IMLA workshop.*

Felty, A.: 1992, 'A Logic Programming approach to implementing higher-order term rewriting'. In: L.-H. Eriksson et al. (eds.): *Second International Workshop on Extensions to Logic Programming*, Vol. 596 of *Lecture Notes in Artificial Intelligence.* pp. 135–61.

Felty, A.: 1993, 'Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language'. *Journal of Automated Reasoning* **11**(1), 43–81.

Felty, A. P.: 2002, 'Two-Level Meta-Reasoning in Coq'. In: V. A. Carreno, C. A. Munoz, and S. Tahar (eds.): *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002.* pp. 198–213.

Gallagher, J. K.: 1993, 'The Use of Proof Plans in Tactic Synthesis'. Ph.D. thesis, University of Edinburgh.

Harper, R., F. Honsell, and G. Plotkin: 1992, 'A Framework for Defining Logics'. *Journal of the ACM* **40**(1), 143–84. Preliminary version in LICS '87.

Hutter, D.: 1990, 'Guiding Inductive Proofs'. In: M. E. Stickel (ed.): *10th International Conference on Automated Deduction*, Vol. 449 of *Lecture Notes in Artificial Intelligence*. pp. 147–161.

Hutter, D.: 1991, 'Pattern-Direct Guidance of Equational Proofs'. Ph.D. thesis, University of Karlsruhe.

Hutter, D.: 1996, 'Using Rippling for Equational Reasoning'. In: S. Hölldoblrt (ed.): *Proceedings 20th German Annual Conference on Artificial Intelligence KI-96*.

Hutter, D. and M. Kohlhase: 2000, 'Managing Structural Information by Higher-Order Colored Unification'. *Journal of Automated Reasoning* **25**(2), 123–164.

Klop, J. W.: 1992, 'Term Rewriting Systems'. In: S. Abramsky, D. Gabbay, and T. S. Maibaum (eds.): *Handbook of Logic in Computer Science, vol 2*, Vol. 2. Oxford: Clarendon Press, pp. 1–116.

Liang, C.: 1992, 'λProlog Implementation of Ripple-Rewriting'. In: *Proceedings of the 1992 Workshop on the λProlog Programming Language*. University of Pennsylvania, Philadelphia, PA, USA.

Luo, Z. and R. Pollack: 1992, 'LEGO Proof Development System: User's Manual'. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh. See also `http://www.dcs.ed.ac.uk/home/lego`.

Miller, D.: 1991, 'A Logic Programming Language with Lambda Abstraction, Function Variables and Simple Unification'. In: P. Schröder-Heister (ed.): *Extensions of Logic Programming*, Vol. 475 of *Lecture Notes in Artificial Intelligence*.

Miller, D.: 2000, 'Abstract syntax for variable binders: An overview'. In: e. a. J. Lloyd (ed.): *Computation Logic (CL'2000)*, Vol. 1861 of *Lecture Notes in Artificial Intelligence*.

Negrete, S. and A. Smaill: 1995, 'Guiding proof search in logical frameworks with rippling'. Technical report, Dept. of Artificial Intelligence, University of Edinburgh.

Nipkow, T.: 1991, 'Higher-order critical pairs'. In: *Proc. 6th IEEE Symp. Logic in Computer Science*. pp. 342–349.

Paulson, L.: 1994, *Isabelle: A Generic Theorem Prover*, Vol. 823 of *Lecture Notes in Computer Science*. Springer.

Richardson, J., A. Smaill, and I. Green: 1998, 'System Description: Proof Planning in Higher-Order Logic with Lambda-Clam'. In: C. Kirchner and H. Kirchner (eds.): *Conference on Automated Deduction (CADE'98)*, Vol. 1421 of *Lecture Notes in Computer Science*. pp. 129–133.

Smaill, A. and I. Green: 1995, 'Automating the synthesis of functional programs'. Research paper 777, Dept. of Artificial Intelligence, University of Edinburgh.

Smaill, A. and I. Green: 1996, 'Higher-order Annotated Terms for Proof Search'. In: J. von Wright, J. Grundy, and J. Harrison (eds.): *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, Vol. 1275 of *Lecture Notes in Computer Science*. Turku, Finland, pp. 399–414. Also available as DAI Research Paper 799.

Yoshida, T., A. Bundy, I. Green, T. Walsh, and D. Basin: 1994, 'Coloured rippling: An extension of a theorem proving heuristic'. In: A. G. Cohn (ed.): *Proceedings of ECAI-94*. pp. 85–89.

Appendix

A. Formal definitions for Embeddings and Annotated Terms

We only consider the case where there is a single-hole under any wave front (or in the embedding situation where there is only one embedding). These are referred to as *simply annotated* terms by Basin & Walsh (Basin and Walsh, 1996) and following them we shall refer to situations where there is only one embedding as *simply embedded*. The presentation for multi-hole wave fronts closely matches the higher-order system of multiple embeddings and transferal of these results to that case is straightforward.

In skeletons we also allow *wild cards* which are used to indicate sinks. As Basin & Walsh point out, the use of sinks is merely to restrict even further the possible ripples, so they may safely be ignored when considering questions of measure.

B. Embeddings

We slightly extend our definition of embeddings to include directional annotations.

DEFINITION 13. *An* embedding*, e, is:*

$$e ::= \texttt{leaf}(pos, dir) | fn(pos, dir, e, [e, \dots, e])$$

where pos is a list of integers and dir is a flag inward outward*.*

We will define a functions *epos* and dir_e on embeddings for extracting the position and direction information of an embedding:

DEFINITION 14. *The embedding position function epos : embedding $\rightarrow$ list(nat) is defined by,*

$$\begin{aligned} epos(\texttt{leaf}(pos, dir)) &= pos & (27)\\ epos(fn(pos, dir, e_1, [e_2, \dots, e_n])) &= pos & (28) \end{aligned}$$

DEFINITION 15. *The embedding direction function dir_e : embedding $\rightarrow$ direction is defined by,*

$$\begin{aligned} dir_e(\texttt{leaf}(pos, dir)) &= dir & (29)\\ dir_e(fn(pos, dir, e_1, [e_2, \dots, e_n])) &= dir & (30) \end{aligned}$$

We extend our definition of a well-embedding from §4 in the obvious way to include direction information.

For convenience we define three projections for extracting the erasure, skeleton and embedding from an embedding expression.

DEFINITION 16. *Let*

$$\begin{aligned} erase_e(e : s \hookrightarrow (t, pos)) &= t & (31) \\ skel_e(e : s \hookrightarrow (t, pos)) &= s & (32) \\ emb(e : s \hookrightarrow (t, pos)) &= e & (33) \end{aligned}$$

C. Wave Fronts and Wave Holes

The concepts of a wave fronts and holes are important. We define them for embeddings as follows:

DEFINITION 17. *An embedding in a well embedded embedding expression, $e : s \hookrightarrow (t, pos)$, is a* wave front *(written $wf_e(e : s \hookrightarrow (t, pos))$) iff $we(e : s \hookrightarrow (t, pos)) \wedge pos \neq epos(e)$*

DEFINITION 18. *An embedding in an well embedded embedding expression, $e : s \hookrightarrow t$, is a* wave hole *(written $(wh_e(e : s \hookrightarrow (t, pos)))$ iff $we(e : s \hookrightarrow t) \wedge pos = epos(e)$*

It should be noted that our definition of a wave hole does not exactly match the traditional one. For instance in the term $f(\boxed{g(\underline{h(x)})})$ our definition makes f, h and x all wave holes whereas the traditional definition makes only h a wave hole.

D. Well Annotated Terms

We make some slight changes to Basin & Walsh's definitions of annotated terms. We've modified their two constructors $wfout$ and $wfin$ into one wf which takes an additional argument *outward* or *inward*.

DEFINITION 19. Well-annotated terms *(or* wats*) are the smallest set such that,*

1. *t is a wat for all unannotated terms;*
2. *$wf(dir, f(t_1, \ldots, t_n)$ is a* wat *iff for at least some i, $t_i = wh(s_i)$ and for each i where $t_i = wh(s_i)$, s_i is a* wat *and for each i where $t_i \neq wh(s_i)$, t_i is an unannotated term;*

3. $f(t_1, \ldots, t_n)$ *is a* wat *where* f *is not* wf *or* wh *iff each* t_i *is a* wat

Since we're only considering the single-holed case we shall treat $skel$ as if it was of type $wats \rightarrow unats$. To make the recursion in later proofs simpler and also provide a case for $wh(t)$ separately so this is a function from $wats \cup wh$ (where wh are expression of the form $wh(t)$ where $wats(t)$) . Our definition is therefore:

DEFINITION 20. *The* skeleton function:$wats \cup wh \rightarrow \mathcal{P}(unats)$ *is defined by*

1. $skel(x) = x$ *for all variables* x*;*

2. $skel(wf(dir, f(t_1, \ldots, t_n))) = s$ *st.* $\exists i.t_i = wh(t'_i) \wedge s = skel(t'_i)\}$

3. $skel(wh(t)) = skel(t)$

4. $skel(f(t_1, \ldots, t_n)) = f(skel(t_1), \ldots, skel(t_n))$ *where* $f \neq wf$ *and* $f \neq wh$*.*

Similarly we extend the definition of the erasure with an extra case for wh

DEFINITION 21. *The* erasure function:$wats \cup wh \rightarrow unats$ *is defined by*

1. $erase(x) = x$ *for all variables* x*;*

2. $erase(wf(dir, f(t_1, \ldots, t_n))) = f(s_1, \ldots, s_n)$ *where if* $t_i = wh(t'_i)$ *then* $s_i = erase(t'_i)$ *else* $s_i = t_i$

3. $erase(wh(t)) = erase(t)$

4. $erase(f(t_1, \ldots, t_n)) = f(s_1, \ldots, s_n)$ *where* $f \neq wf$ *and* $f \neq wh$ *and* $s_i = erase(t_i)$*.*

Since we are in a single hole situation we define a function to return the number of the argument in which a wave hole appears.

DEFINITION 22. *The* wave hole index *(*$whi(wf(dir, f(t_2, \ldots t_n)))$*) is that* i *for which* $t_i = wh(s_i)$*.*

E. Embeddings to Annotated Terms

We provide mappings between well-annotated terms and well-embedded embeddings. These only apply to the first-order case by which we mean terms containing no λ-abstractions and in which no wave fronts appear in the 3rd argument of fn nodes so effectively for $fn(pos, dir, e_1, \bar{e}^n) : s_1(\bar{s}^n) \hookrightarrow (t_1(\bar{t}^n)), pos)$ we can assume that $s_1 = t_1$. All the proofs of the following propositions follow by straightforward induction on the structure of the term t.

DEFINITION 23. *The annotated term to embedding function f_{wte} : $(wats * list(nat) * direction) \rightarrow embedding\_expression$ is defined by,*

$$f_{wte}(c, pos, dir) = \texttt{leaf}(pos, dir) : c \hookrightarrow (c, pos) \tag{34}$$

$$f_{wte}(X, pos, dir) = \texttt{leaf}(pos, dir) : X \hookrightarrow (X, pos) \tag{35}$$

$$\begin{gathered} f_{wte}(t_1(\bar{t}^n), pos, dir) = \\ fn(pos, dir, emb(f_{wte}(t_1, 1 :: pos, dir)), \overline{emb(f_{wte}(t_i, i :: pos, dir))}^{i}) : \\ t_1(\overline{skel_e(f_{wte}(t_i, i :: pos, dir))}^{i}) \hookrightarrow \\ (t_1(\overline{erase_e(f_{wte}(t_i, i :: pos, dir))}^{i}), pos) \end{gathered} \tag{36}$$

$$\begin{gathered} f_{wte}(wf(dir, t_1(\bar{t}^n))), pos, dir') = \\ emb(f_{wte}(t_{whi(t_1(\bar{t}^n))}, whi(t_1(\bar{t}^n)) :: pos, dir)) : \\ skel_e(f_{wte}(t_{whi(t_1(\bar{t}^n))}, whi(t_1(\bar{t}^n)) :: pos, dir)) \hookrightarrow \\ t_1(\overline{erase_e(f_{wte}(t_i, i :: pos, dir))}^{i}), pos) \end{gathered} \tag{37}$$

$$f_{wte}(wh(t)), pos, dir) = f_{wte}(t, pos, dir) \tag{38}$$

PROPOSITION 4.

$$skel_e(f_{wte}(t, pos, dir)) = skel(t) \tag{39}$$

PROPOSITION 5.

$$erase_e(f_{wte}(t, pos, dir)) = erase(t) \tag{40}$$

PROPOSITION 6. *If t is a well-annotated term then*

$$we(f_{wte}(t, pos, dir)) \tag{41}$$

We also define a mapping from well-embeddings to annotated terms and again we can show that this preserves skeletons and erasure and that the result is a *wat*.

DEFINITION 24. *For two position lists p and p' such that $\exists p''.\, p = p' <> p$ (where $<>$ indicates concatenation of lists), $nb(p', p) = hd(p'')$ where $hd(l)$ is the first element of the list l. i.e. nb indicates the* next branch *to be taken in navigating a term tree from position p' in order to reach position p.*

DEFINITION 25. *The embedding to wat function,* $f_{etw} : (embedding\_expression * (wats \to wats \cup wh)) \to wats$ *is defined by,*

$$f_{etw}(\mathtt{leaf}(pos, dir) : c \hookrightarrow (c, pos), h) = h(c) \tag{42}$$

$$f_{etw}(\mathtt{leaf}(pos, dir) : X \hookrightarrow (X, pos), h) = h(X) \tag{43}$$

$$\begin{gathered} epos(e) \neq pos \Rightarrow \\ f_{etw}(e : s \hookrightarrow (t_1(t_2, \dots, t_n)), pos), h) = \\ h(wf(dir_e(e), (t_1(\bar{t_i})[t_{nb(epos(e),pos)})/f_{etw}(e : s \hookrightarrow (t_{nb(epos(e),pos)}, pos), \lambda x.wh(x))]))) \end{gathered} \tag{44}$$

$$\begin{gathered} f_{etw}(fn(pos, dir, e_1, [e_2, \dots, e_n]) : s_1(\bar{s}^n) \hookrightarrow t_1(t_2, \dots, t_n), pos, h) = \\ h(t_1(f_{etw}(e_2 : s_2 \hookrightarrow (t_2, 2 :: pos), \lambda x.x), \dots, f_{etw}(e_n : s_n \hookrightarrow (t_n, n :: pos), \lambda x.x))). \end{gathered} \tag{45}$$

PROPOSITION 7. *If* $e : s \hookrightarrow t$ *is a well-embedding then* $f_{etw}(e : s \hookrightarrow t, \lambda x.x)$ *is a well-annotated term and* $f_{etw}(e : s \hookrightarrow t, \lambda x.wh(x)) = wh(t)$ *where* t *is a well annotated term.*

Proof. by induction on t. □

PROPOSITION 8.

$$skel(f_{etw}(e, f)) = skel_e(e) \tag{46}$$

Proof. By induction on $erase_e(e)$ □

PROPOSITION 9.

$$erase(f_{etw}(tt, \lambda x.x)) = erase_e(tt) \tag{47}$$

Proof. By induction on tt. □

F. The Measure

The definition of Measure from Basin & Walsh is:

DEFINITION 26. *For an annotated term, t, the* out-weight of a position p *(in the skeleton) is the sum of the weights of the (possibly nested) outwards oriented wave-fronts at* p*. The* $in - weight$ *is defined analogously except for inward directed wave-fronts.*

They define the *weight* of a wave front as it's width (i.e the number of function symbols between it and the wave hole). We assume all wave fronts are maximally split and so redefine this as

DEFINITION 27. *The* weight *($\mathcal{M}$) of a wave front is defined as follows:*

$$\begin{aligned} t_1 \neq wf \Rightarrow \mathcal{M}(dir, t_1(t_2, \ldots, t_n)) &= 0 && (48) \\ dir \neq dir' \Rightarrow \mathcal{M}(dir, wf(dir', t)) &= 0 && (49) \\ \mathcal{M}(dir, wf(dir, t)) &= s(\mathcal{M}(dir, t)) && (50) \end{aligned}$$

NB. This only corresponds to Basin & Walsh's definition of weight if we assume that the directions of the wave front can't change between a wave front and a "proper" wave hole. i.e. expression such as $\boxed{f(\underline{\boxed{g(\underline{x})}^{\downarrow}})}^{\uparrow}$ never occur. The embedding formulation can't make this sort of distinction as discussed in §4.

DEFINITION 28. *The* out-measure, *$\mathcal{MO}(t)$ of an annotated term t is a list of length $|t| + 1$ whose i-th element is the sum of out-weights for all term positions in t at depth i. The* in-measure, *$\mathcal{MI}(t)$ is a list whose i-th element is the sum of in-weights for all term positions in t at depth i. The* measure *of an annotated term, $\mathcal{M}(t)$ is the pair of out and in-measures, $< \mathcal{MO}(t), \mathcal{MI}(t) >$*

Note that if for a t at position p in the skeleton of an annotated term, w, $\mathcal{M}(t) = \mathcal{W}_e(f_{wte}(w), 0, p)$, then the definition of measure for both formalisms is the same. A sketch proof of this was provided in §5.