LAYERED PREDICATES*

Flemming Nielson and Hanne Riis Nielson

Computer Science Department, Aarhus University, Bldg. 540
Ny Munkegade,DK-8000 Aarhus C, Denmark

December 1992

Abstract

We review the concept of logical relations and how they interact with structural induction; furthermore we give examples of their use, and of particular interest is the combination with the PER-idea (partial equivalence relations). This is then generalized to Kripke-logical relations; the major application is to show that in combination with the PER-idea this solves the problem of establishing a substitution property in a manner conducive to structural induction. Finally we introduce the concept of Kripke-layered predicates; this allows a modular definition of predicates and supports a methodology of "proofs in stages" where each stage focuses on only one aspect and thus is more manageable. All of these techniques have been tested and refined in "realistic applications" that have been documented elsewhere.

Keywords: logical relations, partial equivalence relations, Kripke-logical relations, layered predicates, Kripke-layered predicates, substitution properties, well-structured proofs, denotational semantics,

*This is a preprint of a paper to appear in the Proceedings of the REX'92 workshop "Semantics - foundations and applications" to be published by Springer Lecture Notes in Computer Science.

correctness of code generation, proof principles.

Contents

0 Introduction

It is common mathematical practice to structure the development of a theory into a series of definitions and a series of facts, lemmas, propositions and theorems. Each definition introduces some concept, predicate or relation. Each fact, lemma, proposition and theorem presents insights of increasing importance and, at least in principle, increasing difficulty of proof. By structuring the main insight into a number of definitions and theorems, the latter structured into a number of lemmas etc., the claim is that each step in the overall development becomes more amenable, easier to conduct and check, and easier to adopt to analogous settings. By combining all the theorems one then obtains the insight desired about the aggregation of concepts introduced.

Well-structured and amenable proofs are of no less importance in computer science than in mathematics. Perhaps they are more important here because the structures studied are often much "bulkier" than in mathematics. As an example consider a realistic programming language with its massive amount

of syntax and syntactic categories; any interesting claim about such a language is likely to be at least as "bulky" as the syntax of the language since every syntactic category must participate in the formulation of the claim and each syntactic construct in the proof. Machine implementations of automatic and semi-automatic proof systems have been devised to help in dealing with such "bulky" proofs but they all need some amount of guidance, e.g. in the form of proof tactics, and some require human interaction to solve subgoals beyond the power of the machine. Thus the need for well-structured proofs is of no less importance for automatic proofs than for manual proofs.

The main problem in adapting the techniques of mathematics to computer science is that most structures in computer science have some higher-order or recursive aspects. Examples include the meaning of procedures, the context free syntax of programming languages, domain equations, and type systems. Here even the definition of the predicates may require some ingenuity and the use of special techniques. As we shall see one such technique is that of logical relations where the predicate is defined to hold on a function if and only if the predicate is preserved across function calls: if the predicate holds of the argument it must also hold of the result. The difficulty now is that preservation of an aggregation of predicates (or a strong predicate) does not imply the preservation of each individual predicate (or of a weaker predicate). The reason is that the strong predicate may be preserved simply because it holds for no arguments and that the weak predicate may fail to be preserved because it holds for no results; we shall study concrete examples in Examples 3.1 and 3.2.

The solution is to be more careful in the aggregation of predicates and to this end a notion of layered predicates is developed. To conduct this development we first give an overview of logical relations and partial equivalence relations and we give examples suggesting that these notions are unavoidable. We next consider Kripke-logical relations and we give an extended example suggesting that this notion is unavoidable for establishing a substitution property. Finally the notion of layered predicates is developed.

Throughout the presentation we aim at avoiding complex examples; how-ever extensive applications of the development may be found in [10, Chapter 6] and in [9] in the context of proving the correctness of code generation. Indeed the present work is a synthesis of the ideas developed there and we aim at presenting the (often rather implicit) ideas in an application-independent

setting. This allows to study the techniques on their own and will make it easier to apply them to other tasks. Of particular interest would be the use of these techniques as a tool in structuring computer-based proofs. Also more general and more mathematical (e.g. categorical) formulations would be an avenue for further research.

1 Logical Relations

Our example language throughout this paper will be a monotyped λ-calculus. Its types t, expressions e, variables v and constants c are given by

$$\begin{array}{lcl} t & ::= & \texttt{num} \mid \texttt{bool} \mid \texttt{charlist} \mid t \to t \\ e & ::= & \lambda v : t.e \mid e\ e \mid v \mid c \\ v & ::= & x \mid y \mid z \mid f \mid g \mid \cdots \\ c & ::= & * \mid + \mid - \mid \cdots \end{array}$$

Here `num`, `bool` and `charlist` are base types and $t_1 \to t_2$ denotes the type of functions from t_1 to t_2. It is merely for simplicity of presentation that sum types, product types and recursive types have not been included. Expressions include λ-abstraction, function application, and the use of variables and constants. We leave the exact nature of variables and constants unspecified and we shall use infix rather than prefix notation for constants.

Example 1.1 To make our examples appear a bit more realistic we shall impose a module structure upon the language. It consists of specifications and implementations. An example specification is

```
spec  sum1:   num → num
with  sum1 maps nonneg.   integers to nonneg.   integers
         /* sum1 x = 1 +...+ x */
```

Here the first line gives the functionality of the function `sum1`. The third line states our intention with the function but this is merely a comment and of no semantic consequence. The second line states a property of `sum1` that any acceptable implementation must satisfy. Part of our work is to formalize the

informal wording of that property but the more difficult part (in general) is to do it in a way that lends itself to proofs by structural induction (on the syntax of implementations). An example implementation then is

$$\texttt{impl sum1 = } \lambda \texttt{x}.\frac{\texttt{x*x}}{2} + \frac{\texttt{x}}{2}$$

Here we take some liberty in the use of the infix notation. We shall shortly return to the proof that the implementation satisfies its specification. □

This language is subject to the usual rules for well-formedness. It is worthwhile to write this out in detail. The judgements are of the form

$$tenv \ \vdash e : t$$

saying that in the type environment $tenv$ the expression e has type t. As usual a type environment is a finite list of pairs of variables and types and we write $tenv(v) = t$ whenever (v, t) is the rightmost pair in $tenv$ of form (v, t') for some t'. The central portions of the inference system are given by

$$\frac{tenv, (v, t) \vdash e : t'}{tenv \vdash \lambda v : t.e : t \to t'}$$

$$\frac{tenv \vdash e_1 : \ t \to t' \ \ tenv \ \vdash e_2 : t}{tenv \vdash e_1 \ e_2 : t'}$$

$$tenv \vdash v : t \quad \text{if } tenv(v) = t$$

$$tenv \vdash c : t \quad \text{if } TYPE(c) = t$$

For constants we have assumed a global assignment $TYPE$ of types to the constants much like the type environment but we shall leave the details implicit. Expressions are uniquely typed; this means that if an expression has two types, as in $tenv \vdash e : t_1$ and $tenv \vdash e : t_2$, then they are equal, i.e. $t_1 = t_2$, but it does not guarantee the existence of a type for all expressions (e.g. the application `1 2`). In examples we shall allow to dispense with the types after λ-abstracted variables. The empty type environment is written ( ).

The semantics of types and expressions is usually parametrized on an interpretation $\mathcal{I}$ of the meaning of base types and constants. An example interpretation is the standard interpretation $\mathbf{S}$ that has

$$\begin{array}{lcll}
\mathbf{S}(\texttt{num}) & = & \text{NUM} & \text{(the flat domain of } \textit{rational} \text{ numbers)} \\
\mathbf{S}(\texttt{bool}) & = & \text{BOOL} & \text{(the flat domain of booleans)} \\
\mathbf{S}(\texttt{charlist}) & = & \text{CHARLIST} & \text{(the flat domain of lists of characters)}
\end{array}$$

The semantics $\mathcal{I}[\![t]\!]$ of a type t is then given by

$$\begin{array}{lcl}
\mathcal{I}[\![\texttt{num}]\!] & = & \mathcal{I}(\texttt{num}) \\
\mathcal{I}[\![\texttt{bool}]\!] & = & \mathcal{I}(\texttt{bool}) \\
\mathcal{I}[\![\texttt{charlist}]\!] & = & \mathcal{I}(\texttt{charlist}) \\
\mathcal{I}[\![t_1 \to t_2]\!] & = & [\mathcal{I}(t_1) \to \mathcal{I}(t_2)]
\end{array}$$

where the arrow constructs the set of total functions and the square brackets extract those that are continuous. It is helpful also to define the semantics $\mathcal{I}[\![tenv]\!]$ of a type environment $tenv$. Here we simply set

$$\mathcal{I}[\![(v_1, t_1), \ldots, (v_n, t_n)]\!] = \mathcal{I}[\![t_1]\!] \times \ldots \times \mathcal{I}[\![t_n]\!]$$

where the righthand side is the n-ary cartesian product ordered componentwise (and the one-point domain if $n = 0$). Corresponding to the notation $tenv(v) = t$ we define the associated projection function

$$\pi_v^{tenv} : \mathcal{I}[\![tenv]\!] \to \mathcal{I}[\![t]\!]$$

but we shall not use space for the tedious details of the formal definition. Turning to expressions we define the semantics $\mathcal{I}[\![e]\!]_{tenv} \in \mathcal{I}[\![tenv]\!] \to \mathcal{I}[\![t']\!]$ of a well-formed expression e, i.e. $tenv \vdash e : t'$, as follows

$$\begin{array}{rcl}
\mathcal{I}[\![\lambda v : t.\ e]\!]_{tenv} & = & \lambda(w_1 \ldots, w_n).\ \lambda w.\mathcal{I}[\![e]\!]_{tenv,(v,t)}(w_1 \ldots, w_n, w) \\
\mathcal{I}[\![e_1\ e_2]\!]_{tenv} & = & \lambda(w_1 \ldots, w_n).\ \mathcal{I}[\![e_1]\!]_{tenv}(w_1 \ldots, w_n) \\
 & & \qquad (\mathcal{I}[\![e_2]\!]_{tenv}(w_1 \ldots, w_n)) \\
\mathcal{I}[\![v]\!]_{tenv} & = & \pi_v^{tenv} \\
\mathcal{I}[\![c]\!]_{tenv} & = & \lambda(w_1 \ldots, w_n).\ \mathcal{I}(c)
\end{array}$$

where the interpretation specifies the meaning $\mathcal{I}(c) \in \mathcal{I}[\![TYPE(c)]\!]$ of constants. We shall not go further into the behaviour of the standard interpretation **S** at this stage.

A predicate over a domain (or set) D is a total function from D to the set $\{true, false\}$ of truth values. We shall write $D \to \{true, false\}$ for the set of predicates over D. An m-ary relation over $D_1, \ldots, D_m$ is simply a predicate over the cartesian product $D_1 \times \ldots \times D_m$ and henceforth we shall regard the words predicate and relation as interchangable.

Definition 1.2 An *indexed relation* R over interpretations $\mathcal{I}_1, \ldots, \mathcal{I}_m$ is a collection of m-ary relations $R_t : \mathcal{I}_1[\![t]\!] \times \ldots \times \mathcal{I}_m[\![t]\!] \to \{true, false\}$ one for each type t. It is a *logical relation* iff

$$R_{t_1 \to t_2}(f_1, \ldots, f_m) \equiv \forall(w_1, \ldots, w_m) : \quad R_{t_1}(w_1, \ldots, w_m) \Downarrow R_{t_2}(f_1(w_1), \ldots, f_m(w_m))$$

holds for all types t_1 and t_2. □

Clearly a logical relation is uniquely determined by its effect on the base types. To avoid excessive use of brackets we write $e \models R[t]$ for $R_t(\mathcal{I}_1[\![e]\!]_{()}( ), \ldots, \mathcal{I}_m[\![e]\!]_{()}( ))$ whenever R is an indexed relation.

We shall next consider some examples of the use of logical relations. These will be grouped into three groups corresponding to increasing complexity of formulation. The fist group is concerned with logical relations over one interpretation only.

Example 1.3 Returning to the `sum1` example our first task is to formalize the interface condition:

```
sum1 maps nonnegative integers to nonnegative integers
```

We may define a logical relation *NONNEG* over **S** as follows:

$$\begin{array}{lcl} NONNEG_{\texttt{num}}(w) & \equiv & w \geq 0 \\ NONNEG_{\texttt{bool}}(w) & \equiv & true \\ NONNEG_{\texttt{charlist}}(w) & \equiv & true \end{array}$$

The definition of $NONNEG_{\texttt{bool}}$ and $NONNEG_{\texttt{charlist}}$ may seem abitrary but they will be instances of a general pattern that will emerge later. In a similar way we may define a logical relation INT as follows:

$$\begin{array}{lcl} INT_{\texttt{num}}(w) & \equiv & w \in \{\ldots,-2,-1,0,1,2,\ldots\} \\ INT_{\texttt{bool}}(w) & \equiv & true \\ INT_{\texttt{charlist}}(w) & \equiv & true \end{array}$$

Given logical relations R' (e.g. $NONNEG$) and R'' (e.g. INT) we may define a logical relation $R' \wedge R''$ by

$$\begin{array}{lcl} (R' \wedge R'')_{\texttt{num}}(w) & \equiv & R'_{\texttt{num}}(w) \wedge R''_{\texttt{num}}(w) \\ (R' \wedge R'')_{\texttt{bool}}(w) & \equiv & R'_{\texttt{bool}}(w) \wedge R''_{\texttt{bool}}(w) \\ (R' \wedge R'')_{\texttt{charlist}}(w) & \equiv & R'_{\texttt{charlist}}(w) \wedge R''_{\texttt{charlist}}(w) \end{array}$$

It is very important to point out, as was already hinted at in the Introduction, that in general $(R' \wedge R'')_t(w)$ will be different from $R'_t(w) \wedge R''_t(w)$; we shall study concrete examples in Examples 3.1 and 3.2. Turning to the logical relation $NONNEG \wedge INT$ we see that

$$\texttt{sum1} \models (NONNEG \wedge INT)[\texttt{num} \to \texttt{num}]$$

is the desired reformulation of the interface condition.

A direct proof would proceed by assuming that `x` is a nonnegative integer and would then show that also $\frac{\texttt{x}*\texttt{x}}{2} + \frac{\texttt{x}}{2}$ is. This is not simply a structural induction because for odd `x` also `x*x` and `x` will be odd; but luckily `x*x+x` is even so that $\frac{\texttt{x}*\texttt{x}}{2} + \frac{\texttt{x}}{2}$ is an integer. The details of this proof are of no interest to us, however.

A more well-structured proof might split the combined proof about the relation $NONNEG \wedge INT$ into separate proofs about $NONNEG$ and INT. Clearly one can show

$$\begin{array}{l} \texttt{sum1} \models NONNEG[\texttt{num} \to \texttt{num}] \\ \texttt{sum1} \models INT[\texttt{num} \to \texttt{num}] \end{array}$$

by the same methods of reasoning that we sketched above. From this we would like to infer

$$\texttt{sum1} \models (NONNEG \wedge INT)[\texttt{num} \to \texttt{num}]$$

We may do so if we have available to us a proof rule

$$\frac{e \models R'[t] \quad e \models R''[t]}{e \models (R' \wedge R'')[t]} \quad \text{if } t \ \ldots$$

but this does not hold for all types t; as was claimed above, the reason is that there are likely to be types t' such that $(R' \wedge R'')_{t'}(w)$ differs from $R'_{t'}(w) \wedge R''_{t'}(w)$. However, the proof rule is available to us for $t = \texttt{num} \to \texttt{num}$ (as well as $t = \texttt{num}$) and the proof carries through. □

Example 1.4 For an example of a somewhat different flavour, and to prepare for the development of the next sections, consider the following module:

```
spec  badge:  charlist → charlist
with  badge only involves 7 bit ASCII characters
impl  badge = λ x.  if # x ≤ 17
                    then "Professor" ++ x
                    else "Prof." ++ x
```

Here # gives the length of a list and ++ concatenates two lists. (In the standard semantics these functions will be strict in each argument.) The purpose of the module is to construct a conference badge but taking into account that some names are long and that badges have fixed sizes. Additionally the badge printer only correctly deals with 7 bit ASCII characters.

To formalize the interface condition we define a logical relation *LOWASCII* by

$$\begin{array}{lcl} LOWASCII_{\texttt{num}}(w) & \equiv & true \\ LOWASCII_{\texttt{bool}}(w) & \equiv & true \\ LOWASCII_{\texttt{charlist}}(w) & \equiv & \text{all characters in } w \text{ have ASCII value} \\ & & \text{at most 127 (or } w = \bot) \end{array}$$

(We shall not bother to be more formal about $LOWASCII_{\texttt{charlist}}$.) This follows the pattern of the previous example. Then

$$\texttt{badge} \models \mathit{LOWASCII}[\texttt{charlist} \rightarrow \texttt{charlist}]$$

presents the desired formalization. Clearly if applied to a name that contains an offending character (like the Danish æ, ø and å) so will the result but at least no such characters will be introduced by `badge` provided that none are present in the argument. □

The second group of examples is concerned with logical relations over a sequence of pairwise distinct interpretations.

Example 1.5 Strictness analysis aims at determining when functions need to evaluate their arguments. It is based upon the 2-point domain

$$\mathbf{2} = \begin{array}{c} \bullet \\ \vdots \\ \bullet \end{array} \begin{array}{l} \top \\ \\ \bot \end{array}$$

An interpretation $\mathbf{I}$ for a simple version of strictness analysis is obtained by specifying

$$\begin{array}{lcl} \mathbf{I}(\texttt{num}) & = & \mathbf{2} \\ \mathbf{I}(\texttt{bool}) & = & \mathbf{2} \\ \mathbf{I}(\texttt{charlist}) & = & \mathbf{2} \end{array}$$

Concerning constants we shall give two examples. If

$$\mathbf{S}(*) = \lambda w_1.\ \lambda w_2. \begin{cases} \bot & \text{if } w_1 = \bot \vee w_2 = \bot \\ w_1 \times w_2 & \text{otherwise} \end{cases}$$

it is natural to set

$$\mathbf{I}(*) = \lambda w_1.\ \lambda w_2.\ w_1 \sqcap w_2$$

where $\sqcap$ denotes binary meet, i.e. $0 \sqcap 0 = 0 \sqcap 1 = 1 \sqcap 0 = 0$ and $1 \sqcap 1 = 1$. Similarly, if

$$\mathbf{S}(\texttt{if}) = \lambda w_1.\ \lambda w_2.\ \lambda w_3. \begin{cases} \bot & \text{if } w_1 = \bot \\ w_2 & \text{if } w_1 = \mathit{true} \\ w_3 & \text{if } w_1 = \mathit{false} \end{cases}$$

then it is natural to set

$$\mathbf{I}(\texttt{if}) = \lambda w_1.\ \lambda w_2.\ \lambda w_3.\ \begin{cases} \bot & \text{if } w_1 = \bot \\ w_2 \sqcup w_3 & \text{otherwise} \end{cases}$$

where $\sqcup$ denotes binary join.

To express the correctness of the strictness analysis we define a logical relation COR over the interpretations $\mathbf{S}$ and $\mathbf{I}$. It is given by

$$\begin{array}{lcl} COR_{\texttt{num}}(w^1, w^2) & \equiv & (w^2 = \bot \Rightarrow w^1 = \bot) \\ COR_{\texttt{bool}}(w^1, w^2) & \equiv & (w^2 = \bot \Rightarrow w^1 = \bot) \\ COR_{\texttt{charlist}}(w^1, w^2) & \equiv & (w^2 = \bot \Rightarrow w^1 = \bot) \end{array}$$

It is then quite standard to prove the correctness of $*$ and `if`, but we shall dispense with the details. □

The third group of examples is concerned with logical relations over a sequence of interpretations that are not necessarily pairwise distinct.

Example 1.6 The need for more than one appearance of the same interpretation arises when we want to relate values rather than just express properties about them. To be more specific let us consider the following module:

```
spec sq:  num → num
with sq is independent of the sign of the argument
impl sq = λx. x * x
```

It is not difficult to be more formal about the interface condition. Some possibilities are

$$\begin{array}{ll} \texttt{sq} = \texttt{sq} \circ abs & \\ \texttt{sq} = \texttt{sq} \circ neg & \\ \forall x^1, x^2 : abs(x^1) = abs(x^2) \Rightarrow {}_{\text{sq}}(x^1) = {}_{\text{sq}}(x^2) & \text{(a)} \end{array}$$

where abs is the function that gives the absolute value and neg is the function that multiplies by (-1).

However, these formulations do not immediately lend themselves to proof by structural induction. As we shall see below this will be the case if we can use the framework of logical relations. Our first approach will be to consider

$$\forall x^1, x^2 : abs(x^1) = abs(x^2) \Rightarrow abs(\mathtt{sq}(x^1)) = abs(\mathtt{sq}(x^2) \qquad \text{(b)}$$

We may define a logical relation SQ over interpretations $\mathbf{S}$, $\mathbf{S}$ by

$$\begin{aligned} SQ_{\mathtt{num}}(w^1, w^2) &\equiv abs(w^1) = abs(w^2) \\ SQ_{\mathtt{bool}}(w^1, w^2) &\equiv w^1 = w^2 \\ SQ_{\mathtt{charlist}}(w^1, w^2) &\equiv w^1 = w^2 \end{aligned}$$

and clearly (b) is then equivalent to

$$\mathtt{sq} \models SQ[\mathtt{num} \rightarrow \mathtt{num}]$$

Since `sq` always yields a nonnegative result this is equivalent to (a) as well.

To capture (a) directly we may proceed as follows. Rather than having the base type `num` we shall have several distinct versions; for the present purposes $\mathtt{num}_{abs}$ and $\mathtt{num}_{id}$ suffice. These will be interpreted in the same way in all interpretations but their presence allows us to give the following modified definition of the logical relation SQ over $\mathbf{S}$, $\mathbf{S}$:

$$\begin{aligned} SQ_{\mathtt{num}_{abs}}(w^1, w^2) &\equiv abs(w^1) = abs(w^2) \\ SQ_{\mathtt{num}_{id}}(w^1, w^2) &\equiv w^1 = w^2 \\ SQ_{\mathtt{bool}}(w^1, w^2) &\equiv w^1 = w^2 \\ SQ_{\mathtt{charlist}}(w^1, w^2) &\equiv w^1 = w^2 \end{aligned}$$

Then the condition

$$\mathtt{sq} \models SQ[\mathtt{num}_{abs} \rightarrow \mathtt{num}_{id}]$$

is equivalent to the desired interface condition (a).

For any base type t, i.e. $t \in \{\mathtt{num}_{abs}, \mathtt{num}_{id}, \mathtt{bool}, \mathtt{charlist}\}$, each SQ_t is an equivalence relation over $\mathbf{S}[\![t]\!]$ i.e. SQ_t is a reflexive, transitive and symmetric

relation. But is it more advantageous to exploit only the weaker fact that each such SQ_t is a *partial equivalence relation* (abbreviated PER); this just means that for a base type t each SQ_t is a (not necessarily reflexive) transitive and symmetric relation. Then one can show by structural induction over all types t that SQ_t is a partial equivalence relation over $\mathbf{S}[\![t]\!]$, i.e. we do not need to restrict our attention to base types only.(This would not be the case if we had studied equivalence relations.)

So our approach is essentially that of [1]. A notational difference is that our formulation is close to that of the *faithfulness* relation studied in [8]. This means that the formulation would be useful also for analyses carried out by means of non-standard type inference. The use of binary relations over the same domain, e.g. in the form of partial equivalence relations, rather than just unary predicates seems to be necessary for validating a number of analyses. Examples include binding time analysis [2] and liveness analysis [6]. □

Logical relations are useful because they interact very well with the semantics of the λ-calculus. In particular they are well suited to proofs by structural induction.

Definition 1.7 An indexed relation R over $\mathcal{I}_1, \ldots, \mathcal{I}_m$ is said to *admit structural induction* whenever it satisfies the following condition:

> For every type environment $tenv = (v_1, t_1), \ldots, (v_n, t_n)$ and every well-typed expression e of type t, i.e. $tenv \vdash e : t$; if
>
> $R_{t_i}(w_i^1, \ldots, w_i^m)$ for all $i = 1, \ldots, n$
>
> $R_{t'}(\mathcal{I}_1(c'), \ldots, \mathcal{I}_m(c'))$ for all constants c' of type t' occurring in e
>
> then
>
> $R_t(\mathcal{I}[\![e]\!]_{tenv}(w_1^1, \ldots, w_n^1), \ldots, \mathcal{I}[\![e]\!]_{tenv}(w_1^m, \ldots, w_n^m)$ □

Lemma 1.8 *Logical relations admit structural induction.* □

Proof Using the notation of the definition this is a structural induction over e. For a variable v the result is immediate from the assumptions. This is also the case for a constant c. For an application $e_1\ e_2$ we use the induction hypothesis to obtain

$$R_{t'\to t}(\mathcal{I}_1[\![e_1]\!]_{tenv}(w_1^1,\ldots,w_n^1),\ \ldots\ ,\mathcal{I}_m[\![e_1]\!]_{tenv}(w_1^m,\ \ldots\ ,w_n^m))$$
$$R_{t'}(\mathcal{I}_1[\![e_2]\!]_{tenv}(w_1^1,\ldots,w_n^1),\ \ldots\ ,\mathcal{I}_m[\![e_2]\!]_{tenv}(w_1^m,\ \ldots\ ,w_n^m))$$

and then we use that R is logical; to be more specific we use the following proof rule

$$\frac{R_{t'\to t}(f_1,\ldots,f_m)\quad R_{t'}(w_1,\ldots,w_m)}{R_t(f_1(w_1),\ldots,f_m(w_m))}$$

whose validity is a direct consequence of R being logical. For a λ-abstraction $\lambda v{:}t'.\ e$ we must show

$$R_{t'\to t''}(\mathcal{I}_1[\![\lambda v{:}t'.\ e]\!]_{tenv}(w_1^1,\ldots,w_n^1),\ldots,\mathcal{I}_m[\![\lambda v{:}t'.\ e]\!]_{tenv}(w_1^m,\ldots,w_n^m))$$

By R being logical this amounts to assuming

$$R_{t'}(w_{n+1}^1,\ldots,w_{n+1}^m)$$

and showing

$$R_{t''}(\mathcal{I}_1[\![\lambda v{:}t'.\ e]\!]_{tenv}(w_1^1,\ldots,w_n^1)(w_{n+1}^1),\ldots,\ \mathcal{I}_m[\![\lambda v{:}t'.\ e]\!]_{tenv}(w_1^m,\ldots,w_n^m)(w_{n+1}^m)).$$

But since

$$\mathcal{I}_i[\![\lambda v{:}t'.\ e]\!]_{tenv}(w_1^i,\ldots,w_n^i)(w_{n+1}^i)=\mathcal{I}_i[\![e]\!]_{tenv,(v,t')}(w_1^i,\ldots,w_n^i,w_{n+1}^i).$$

this follows from the induction hypothesis. □

Taking $n=0$ and $m=1$ we get:

Corollary 1.9 If the closed expression e has type t, i.e. $(\ )\vdash e:t$, and $c'\models R[t']$ for all constants c' of type t' occurring in e, then $e\models R[t]$ provided that R is logical. □

Historical Remark The concept of logical relations, including the result

on structural induction (our Lemma 1.8), is often attributed to [12]. Actually, [11] predated [12] and contained many of the ideas and one should also acknowledge the relational functors of [13]. That binary relations over a set, rather than just unary predicates, is sometimes needed for abstract interpretation was first realized in [6] in terms of a distinction between "first-order" and "second-order" analyses; the link to partial equivalence relations is due to [1] .

2 Kripke-Logical Relations

The definitions of the predicates of the previous section were mostly rather natural. However, in Example 1.6 the use of partial equivalence relations accounted for a somewhat different flavour of the formulation. To motivate the development of the present section it is worthwhile to look closer at this difference.

All but one of the examples of the previous section were such that the formulation of the predicates was quite natural at level 0, i.e. for base types. The extension to higher levels, i.e. function types, was then accomplished using the general technique of logical relations. In Example 1.6 the predicate was quite natural to formulate at level 1, i.e. for functions in `num` $\to$ `num`. The formulation at level 0, i.e. for `num`, was not so straightforward and we found it necessary to use partial equivalence relations, or more precisely, to use the same interpretation more than once. Then the extension to higher levels could be accomplished using the general technique of logical relations.

In this section we shall consider an example that is more along the lines of Example 1.6 than the other examples. But it presents additional complications that we shall solve by parameterizing the predicates with elements drawn from a partially ordered set.

Example 2.1 As a variation of our `badge` example consider the following module for writing a letter of invitation:

```
spec letter: charlist → charlist
with letter  is a character list with hole(s) in it
impl letter  = λx. "Dear Professor" ++ x ++", We hereby
```

`invite ... "`

By mapping `letter` onto a list of names we will then obtain a list of letters. To ensure that all letters are materially the same, e.g. any discount is offered uniformly to all recipients, we request that `letter` is a character string with hole(s) in it; these holes will be filled with the names of recipients. To be more formal we may rephrase this as

$$\exists l_0, \ldots, l_n : \texttt{letter} = \lambda \texttt{x}.\ (l_0 \texttt{++x++} l_1 \texttt{++} \ldots \texttt{++x++} l_n)$$

Similar situations arise frequently in correctness proofs for code generation based on denotational semantics [9, 10].

The above formulation is given at level 1, i.e. on functions in `charlist` $\rightarrow$ `charlist`, and so we need to find a definition at level 0. Trying to adapt the use of partial equivalence relations we might search for a relation $\sim$ and rephrase the interface condition as follows:

$$\texttt{x} \sim \texttt{y} \Rightarrow \texttt{letter x} \sim \texttt{letter y}$$

However, this approach does not seem to work because of difficulties in defining a relation like $\sim$. The problem is that when analyzing the result of `letter x` it is not possible to distinguish between those substrings that are present because of the insertion of the parameter, and those substrings that just happened to be part of $l_0, \ldots, l_n$.

To overcome this problem we shall pretend that there are special characters called *holes* that may be used to indicate where substitutions should occur. We shall write $l = l_1[l_2/h]$ whenever l is obtained from l_1 by replacing each hole h by the substring l_2. (A formal definition will be given shortly.) The interface condition will then be formulated as follows:

$$SUBST(\{(h, w)\})([h], w) \Rightarrow SUBST(\{(h, w)\})(\texttt{letter}[h], \texttt{letter}\ w)$$

where the predicate is given by

$$SUBST(\{(h, l)\})(l_1, l_2) \equiv (l_1[l/h] = l_2)$$

and $[h]$ denotes a one-element list consisting of the character denoted by h. From this we shall see that we can infer $\texttt{letter}\ w = (\texttt{letter}[h])[w/h]$ and that the interface condition then holds. The formal details follow. □

Example 2.2 Before going into the formal development it is worthwhile to investigate the number of holes that will be necessary. So far we have only seen the need for one but consider the following module:

```
spec invite:  charlist → charlist → charlist
with invite is a character list with hole(s) in it
impl invite   = λy. λx. "Dear Mr.   "++ x ++",
                        You are hereby invited
                        to the "++ x ++" conference, ... "
```

(This might be suitable for a conference bureau.) To formulate the interface condition properly we will need two different holes and in general we will need an arbitrary finite number of holes. This means that the parameter to the *SUBST* relation will be a finite set and not just a single ton set; these parameters are naturally ordered by subset inclusion.—In terms of the motivating applications of code generation, this phenomenon arises whenever nested fixed points (e.g. nested while loops) are allowed. □

In the remainder of this section we develop the general theory of Kripke-logical relations and study some of their properties. We then go on to the rather more demanding task of applying Kripke-logical relations to Example 2.1.

Definition 2.3 A parameterized and indexed relation R over a non-empty partially ordered set Δ and interpretations $\mathcal{I}_1, \ldots, \mathcal{I}_m$ is a collection of parameterized m-ary relations $R_t[\delta] \ : \ \mathcal{I}_1[\![t]\!] \times \ldots \times \mathcal{I}_m[\![t]\!] \to \{true, false\}$ one for each type t and $\delta \in \Delta$. It is a *Kripke-indexed relation* iff

$$\forall \delta' \sqsupseteq \delta : R_t[\delta](w_1, \ldots, w_m) \Rightarrow R_t[\delta'](w_1, \ldots, w_m)$$

holds for all types t. It is a *Kripke-logical relation* iff it is a Kripke-indexed relation and

$$R_{t_1\to t_2}[\delta](f_1,\ldots,f_m) \equiv \forall\delta' \sqsupseteq \delta : \forall(w_1,\ldots,w_m) : \\ R_{t_1}[\delta'](w_1,\ldots,w_m) \Rightarrow R_{t_2}[\delta'](f_1(w_1),\ldots,f_m(w_m))$$

holds for all types t_1 and t_2. □

It is possible to weaken the assumptions on Δ, e.g. to be a quasi-ordered set, but for our purposes this is hardly worth the effort. Clearly a Kripke-logical relation is uniquely determined by its effect on the base types. A logical relation may be regarded as a Kripke-logical relation with Δ having only one element. The need for Δ to have more than one element was hinted at in Example 2.2. The need for the "$\forall\delta' \sqsupseteq \delta$:" will be illustrated in the proof of Lemma 2.13. To avoid excessive use of semantic brackets we shall write $e \models R[t,\delta]$ for $R_t[\delta](\mathcal{I}_1[\![e]\!]_{(\ )}(\ ),\ldots,\mathcal{I}_m[\![e]\!]_{(\ )}(\ ))$ and $e \models R[t]$ for $\forall\delta\in\Delta : e \models R[t,\delta]$ whenever R is a Kripke-indexed relation.

Fact 2.4 If R is a Kripke-indexed relation and Δ has a least element, $\bot_\Delta$, then $e \models R[t,\bot_\Delta]$ is equivalent to $e \models R[t]$. □

Luckily Kripke-logical relations share the good properties of logical relations, namely that they are well suited to proofs by structural induction.

Definition 2.5 A Kripke-indexed relation R over Δ and $\mathcal{I},\ldots,\mathcal{I}_m$ is said to *admit structural induction* whenever it satisfies the following condition:

> For every type environment $tenv = (v_1,t_1),\ldots,(v_n,t_n)$,
> every element $\delta\in\Delta$ and every well-typed expression e of type t,
> i.e. $tenv \vdash e : t$; if
>
> $R_{t_i}[\delta](w_i^1,\ldots,w_i^m)$ for all $i = 1,\ldots,n$
> $R_{t'}[\delta](\mathcal{I}(c'),\ldots,\mathcal{I}_m(c'))$ for all constants c' of type t' occurring in e
>
> then
> $R_t[\delta](\mathcal{I}_1[\![e]\!]_{tenv}(w_1^1,\ldots,w_n^1),\ldots,\mathcal{I}_m[\![e]\!]_{tenv}(w_1^m,\ldots,w_n^m))$ □

Lemma 2.6 *Kripke-logical relations admit structural induction.* □

Proof. Using the notation of the definition this is a structural induction over e. For a variable v the result is immediate from the assumptions. This is also the case for a constant c. For an application $e_1\ e_2$ we use the induction hypothesis to obtain

$$R_{t'\to t}[\delta](\mathcal{I}_1[\![e_1]\!]_{tenv}(w_1^1,\ldots,w_n^1),\ldots,\mathcal{I}_m[\![e_1]\!]_{tenv}(w_1^m,\ldots,w_n^m))$$
$$R_{t'}[\delta](\mathcal{I}_1[\![e_2]\!]_{tenv}(w_1^1,\ldots,w_n^1),\ldots,\mathcal{I}_m[\![e_2]\!]_{tenv}(w_1^m,\ldots,w_n^m))$$

and then we use that R is Kripke-logical; to be more specific we use the following proof rule

$$\frac{R_{t'\to t}[\delta](f_1,\ldots,f_m) \quad R_{t'}[\delta](w_1,\ldots,w_m)}{R_t[\delta](f_1(w_1),\ldots,f_m(w_m))}$$

whose validity is a direct consequence of R being Kripke-logical (and $\delta \sqsupseteq \delta$). For a λ-abstraction $\lambda v : t'.\ e$ we must show

$$R_{t'\to t}[\delta](\mathcal{I}_1[\![\lambda v : t'.\ e]\!]_{tenv}(w_1^1,\ldots,w_n^1),\ldots,\mathcal{I}_m[\![\lambda v : t'.\ e]\!]_{tenv}(w_1^m,\ldots,w_n^m))$$

By R being Kripke-logical this amounts to choosing $\delta' \sqsupseteq \delta$ and assuming

$$R_{t'}[\delta'](w_{n+1}^1,\ldots,w_{n+1}^m)$$

and showing

$$R_{t'}[\delta'](\mathcal{I}_1[\![\lambda v : t'.\ e]\!]_{tenv}(w_1^1,\ldots,w_n^1)(w_{n+1}^1),\ldots,$$
$$\mathcal{I}_m[\![\lambda v : t'.\ e]\!]_{tenv}(w_1^m,\ldots,w_n^m)(w_{n+1}^m))$$

But by assumption we have

$$R_{t_i}[\delta](w_i^1,\ldots,w_i^m) \text{ for all } i = 1,\ldots,n$$

so using the proof rule

$$\frac{R_t[\delta](w_1,\ldots,w_m)}{R_t[\delta'](w_1,\ldots,w_m)} \ \delta' \sqsupseteq \delta$$

we obtain

$$R_{t_i}[\delta'](w_i^1,\ldots,w_i^m) \text{ for all } i = 1,\ldots,n$$

As we also have

$$R_{t_i}[\delta'](w_i^1, \ldots, w_i^m) \text{ for all } i = n+1$$

it follows from the induction hypothesis that

$R_t[\delta'](\mathcal{I}_1[\![e]\!]_{tenv,(v,t')}(w_1^1, \ldots, w_n^1, w_{n+1}^1), \ldots, \mathcal{I}_m[\![e]\!]_{tenv,(v,t')}(w_1^m, \ldots, w_n^m, w_{n+1}^m))$

But since

$$\mathcal{I}_i[\![\lambda v : t'.\, e]\!]_{tenv}(w_1^i, \ldots, w_n^i)(w_{n+1}^i) = \mathcal{I}_i[\![e]\!]_{tenv,(v,t')}(w_1^i, \ldots, w_n^i, w_{n+1}^i))$$

this is the desired result. □

Taking $n = 0$ and $m = 1$ we get:

Corollary 2.7 If the closed expression e has type t, i.e. $(\ ) \vdash e : t$, and $c' \models R[t']$ for all constants c' of type t' occurring in e, then $e \models R[t]$ whenever R is Kripke-logical. □

Extended Example: Establishing Substitution Properties

To apply the technique of Kripke-logical relations to our `letter` example we must take care of a few formalities before defining the *SUBST* relation. We begin by looking closer at the standard interpretation **S** and in particular the equation

$$\mathbf{S}(\texttt{charlist}) = \text{CHARLIST}$$

where CHARLIST is the flat domain of lists of characters. Henceforth we shall assume that

$$\text{CHARLIST} = ((\text{NORMAL} \cup \text{HOLE})^*)_\perp$$

where NORMAL is a (finite) set of characters, HOLE is a disjoint (and in general infinite) set of "hole characters", $(-)^*$ constructs lists and $(-)_\perp$

constructs flat domains. We shall assume that NORMAL includes all the usual ASCII characters and we shall write $h_1, h_2, \ldots$ for the elements of HOLE.

The partially ordered set Δ has as elements those sets of pairs of holes and lists of characters that are acceptable according to the definition given below. The partial order is subset inclusion and it will emerge that $\emptyset$, the empty set, is the least element. A set δ is *acceptable* iff

- $\delta \subseteq \text{HOLE} \times (\text{NORMAL} \cup \text{HOLE})^*$,
- δ is finite,
- δ is functional, i.e. $(h, l_1), (h, l_2) \in \delta \Rightarrow l_1 = l_2$,
- $(h, l) \in \delta \Rightarrow \text{FH}(l) \subseteq \text{DOM}(\delta)$

where $\text{FH}(l)$ is the set of free holes in l, i.e.

$$\begin{aligned} \text{FH}([\,]) &= \emptyset \\ \text{FH}([c]\texttt{++}l) &= \begin{cases} \{c\} \cup \text{FH}(l) & \text{if } c \in \text{HOLE} \\ \text{FH}(l) & \text{if } c \in \text{NORMAL} \end{cases} \end{aligned}$$

and $\text{DOM}(\delta)$ is the "domain" of δ, i.e.

$$\text{DOM}(\delta) = \{h \mid \exists l : (h, l) \in \delta\}$$

The first three conditions for acceptability are rather straightforward; the fourth condition is of a more technical nature and is suitable for the subsequent development (see e.g. Fact 2.10). Whenever δ is acceptable and $h \in \text{DOM}(\delta)$ we shall write $\delta(h)$ for the unique l such that $(h, l) \in \delta$.

We write $l[l_1/h_1, \ldots, l_m/h_m]$ for the result of substituting each list l_i for each hole h_i in l. More formally we have

$$\begin{aligned} [\,][l_1/h_1, \ldots, l_m/h_m] &= [\,] \\ ([c]\texttt{++}l)[l_1/h_1, \ldots, l_m/h_m] &= \begin{cases} [c]\texttt{++}(l[l_1/h_1, \ldots, l_m/h_m]) \\ \quad \text{if } c \notin \{h_1, \ldots, h_m\} \\ l_i\texttt{++}(l[l_1/h_1, \ldots, l_m/h_m]) \\ \quad \text{if } c = h_i \end{cases} \end{aligned}$$

and it is convenient to write also $\bot[l_1/h_1, \ldots, l_m/h_m] = \bot$ as well as $\mathrm{FH}(\bot) = \emptyset$.

We may then attempt to define a Kripke-logical relation $SUBST$ over Δ and **S**,**S** as follows:

$$\begin{aligned}
&SUBST_{\texttt{num}}[\delta](w^1, w^2) \equiv (w^1 = w^2)\\
&SUBST_{\texttt{bool}}[\delta](w^1, w^2) \equiv (w^1 = w^2)\\
&SUBST_{\texttt{charlist}}[\{(h_1, l_1), \ldots (h_m, l_m)\}](w^1, w^2) \equiv\\
&\qquad\qquad (w^1[l_1/h_1, \ldots, l_m/h_m] = w^2) \wedge\\
&\qquad\qquad (\mathrm{FH}(w^1) \subseteq \{h_1, \ldots, h_m\})
\end{aligned}$$

Fact 2.8 For all base types t, if $SUBST_t[\delta](w^1, w^2)$ then $(w^1 = \bot) \vee (w^2 = \bot)$ is equivalent to $w^1 = \bot = w^2$. □

Proof. For $t = \texttt{num}$ and $t = \texttt{bool}$ the result is trivial. For $t = \texttt{charlist}$ it follows because $\delta \in \Delta$ implies that any $(h, l) \in \delta$ has $l \neq \bot$. (It is unlikely that the result holds for function types.) □

Lemma 2.9 *SUBST as defined above is a Kripke-logical relation.* □

Proof. It suffices to prove that

$$\delta' \sqsupseteq \delta \wedge\ SUBST_t[\delta](w^1, w^2) \Rightarrow SUBST_t[\delta'](w^1, w^2)$$

for all base types $t \in \{\texttt{num}, \texttt{bool}, \texttt{charlist}\}$. This is immediate except for $t = \texttt{charlist}$. So assume that $\delta' \sqsubseteq \delta$ and that $SUBST_{\texttt{charlist}}[\delta](w^1, w^2)$. If $w^1 = \bot$ or $w^2 = \bot$ then $w^1 = \bot = w^2$ and the result is immediate so we shall henceforth assume that $w^1 \neq \bot$ and $w^2 \neq \bot$. It is possible to find $m' \geq m$ and $h_1, l_1, \ldots, h_{m'}, l_{m'}$ such that

$$\begin{aligned}
&\delta = \{(h_1, l_1), \ldots (h_m, l_m)\}\\
&\delta' = \{(h_1, l_1), \ldots (h_{m'}, l_{m'})\}
\end{aligned}$$

Our assumption

$$SUBST_{\texttt{charlist}}[\delta](w^1, w^2)$$

implies that

$$\mathrm{FH}(w^1) \subseteq \{h_1, \ldots, h_m\}$$

and from this

$$\mathrm{FH}(w^1) \subseteq \{h_1, \ldots, h_{m'}\}$$

immediately follows since $m' \geq m$. Next our assumption

$$SUBST_{\texttt{charlist}}[\delta](w^1, w^2)$$

also implies that

$$w^1[l_1/h_1, \ldots, l_m/h_m] = w^2$$

and from this

$$w^1[l_1/h_1, \ldots, l_{m'}/h_{m'}] = w^2$$

follows because we also know that

$$\mathrm{FH}(w^1) \subseteq \{h_1, \ldots, h_m\}$$

This establishes $SUBST_{\texttt{charlist}}[\delta'](w^1, w^2)$.

It is convenient to note also

Fact 2.10 If $SUBST_{\texttt{charlist}}[\{(h_1, l_1), \ldots, (h_m, l_m)\}](w^1, w^2)$ then

$$\mathrm{FH}(w^1) \subseteq \{h_1, \ldots, h_m\}$$
$$\mathrm{FH}(w^2) \subseteq \bigcup_{i=1}^{m} \mathrm{FH}(l_i) \subseteq \{h_1, \ldots, h_m\}$$ □

Proof. The first condition is immediate and then gives the first inclusion in the second condition because $w^1[l_1/h_1, \ldots, l_m/h_m] = w^2$. The second inclusion in the second condition then follows from acceptability of $\{(h_1, l_1), \ldots, (h_m, l_m)\}$ □

We now have two tasks ahead of us. One is to show that our $SUBST$ relation is satisfied by `letter` and the other is to show that the $SUBST$ relation does express the substitution property that we are interested in.
We begin with the former.

Lemma 2.11 $\texttt{letter} \models SUBST[\texttt{charlist} \rightarrow \texttt{charlist}]$. □

For the proof we consider $\delta \in \Delta$. To show the result we must consider $\delta' \sqsupseteq \delta$ and assume that

$$SUBST_{\texttt{charlist}}[\delta'](w^1, w^2)$$

and then show

$$SUBST_{\texttt{charlist}}[\delta'](\texttt{letter}\ w^1, \texttt{letter}\ w^2)$$

Writing

$$\delta' = \{(h_1, l_1), \ldots (h_m, l_m)\}$$

our assumptions amount to

$$\mathrm{FH}(w^1) \subseteq \{h_1, \ldots, h_m\}$$

and

$$w^1[l_1/h_1, \ldots, l_m/h_m] = w^2$$

Since

```
letter       = λx. "Dear Professor   "++ x ++",
                   We hereby invite ... "
```

and `'D','e',`... ∈ NORMAL it follows that

$$\mathrm{FH}(\texttt{letter}\ w^1) = \mathrm{FH}(w^1) \subseteq \{h_1, \ldots, h_m\}$$

$$\begin{aligned}(\texttt{letter}\ w^1)[l_1/h_1, \ldots, l_m/h_m] \\ &= \texttt{letter}(w^1[l_1/h_1, \ldots, l_m/h_m]) \\ &= \texttt{letter}\ w^2\end{aligned}$$

and this establishes the desired result.

Example 2.12 To show that the *SUBST* relation is not trivially true we shall show that

$$\neg(\texttt{badge} \models \mathit{SUBST}[\texttt{charlist} \to \texttt{charlist}])$$

First define

$$\begin{aligned}l_1 &= \texttt{"this is a name with 39 characters in it"} \\ \delta &= \{(h_1, l_1)\}\end{aligned}$$

and note that $\delta \in \Delta$ and clearly $\mathit{SUBST}_{\texttt{charlist}}[\delta]([h_1], l_1)$. But

$$\begin{aligned}\texttt{badge}\ [h_1] &= \texttt{"Professor "++}\ [h_1] \\ \texttt{badge}\ l_1 &= \texttt{"Prof. "++}\ l_1\end{aligned}$$

and clearly $\mathit{SUBST}_{\texttt{charlist}}[\delta](\texttt{badge}[h_1], \texttt{badge}\ l_1)$ fails. □

That *SUBST* expresses the desired substitution property follows from

Lemma 2.13

$$\begin{array}{l}\mathit{SUBST}_{\texttt{charlist}\to\texttt{charlist}}[\emptyset](f, f) \wedge f \neq \bot \\ \Updownarrow \\ \exists w_0, \ldots, w_n \in \mathrm{CHARLIST}\ : \forall w \in \mathrm{CHARLIST} \setminus \{\bot\} : \\ \quad (f(w) = w_0\texttt{++}w\texttt{++}w_1\texttt{++}\ldots\texttt{++}w\texttt{++}w_n) \wedge \\ \quad (\forall i : \mathrm{FH}(w_i) = \emptyset \wedge w_i \neq \bot)\end{array}$$

Proof. The downward implication is the more interesting one. The proof proceeds in three stages:

Stage 1 Choose some $h \in \mathrm{HOLE}$ (e.g. the one with minimal index). Define $w_0, \ldots, w_n \in \mathrm{CHARLIST}$ by the conditions that

$$f[h] = w_0 \texttt{++} [h] \texttt{++} w_1 \texttt{++} \ldots \texttt{++} [h] \texttt{++} w_n$$
$$\forall i : h \notin \mathrm{FH}(w_i) \wedge w_i \neq \bot)$$

This is possible if $f[h] \neq \bot$ and then uniquely defines $w_0, \ldots, w_n$ (subject to the choice of h).

To see that $f[h] \neq \bot$ suppose by way of contradiction that $f[h] = \bot$. Since $f \neq \bot$ there exists $w \in \mathrm{CHARLIST}$ such that $f\ w \neq \bot$; we may without loss of generality assume that $w \neq \bot$. Now define

$$\delta = \{(h, w)\} \cup \bigcup \{(h', [h']) \mid h' \in \mathrm{FH}(w) \wedge h' \neq h\}$$

and note that $\delta \in \Delta$. Clearly $SUBST_{\texttt{charlist}}[\delta]([h], w)$ and since $\delta \sqsupseteq \emptyset$, we get $SUBST_{\texttt{charlist}}[\delta](f[h], f\ w)$. But since $f[h] = \bot \neq f\ w$, this is a contradiction (Fact 2.8).

Stage 2 Let $w \in \mathrm{CHARLIST}$ be given such that $w \neq \bot$. As in Stage 1 define

$$\delta = \{(h, w)\} \cup \bigcup \{(h', [h']) \mid h' \in \mathrm{FH}(w) \wedge h' \neq h\}$$

and obtain that $SUBST_{\texttt{charlist}}[\delta](f[h], f\ w)$. This means that

$$\begin{aligned} f\ w &= (f[h])[w/h] \\ &= w_0 \texttt{++} [h] \texttt{++} w_1 \texttt{++} \ldots \texttt{++} [h] \texttt{++} w_n[w/h] \\ &= w_0 \texttt{++} w \texttt{++} w_1 \texttt{++} \ldots \texttt{++} w \texttt{++} w_n \end{aligned}$$

as was to be shown.

Stage 3 To show the remaining properties of $w_0, \ldots, w_n$ we first define

$$\delta = \{(h, [\,])\}$$

and note that $\delta \in \Delta$. Since $SUBST_{\texttt{charlist}}[\delta]([h], [\,])$, it follows from the assumptions that $SUBST_{\texttt{charlist}}[\delta](f[h], f[\,])$. Using Fact 2.10 we have $\mathrm{FH}(f[\,]) \subseteq \emptyset$ and by Stage 2 (with $w = [\,]$) this gives $\forall i : \mathrm{FH}(w_i) = \emptyset$ as desired.

The upward implication is along the lines of the proof of Lemma 2.11. We provide the details by means of:

Stage 4 Suppose that $w_0, \ldots, w_n$ are chosen such that

$$\forall w \neq \bot : f\ w = w_0 \texttt{++} w \texttt{++} w_1 \texttt{++} \ldots \texttt{++} w \texttt{++} w_n$$
$$\forall i\colon \mathrm{FH}(w_i) = \emptyset \wedge w_i \neq \bot$$

Then clearly $f \neq \bot$. To show $SUBST_{\texttt{charlist}\to\texttt{charlist}}[\emptyset](f, f)$ take $\delta \in \Delta$ such that $SUBST_{\texttt{charlist}}[\delta](w^1, w^2)$ and show $SUBST_{\texttt{charlist}}[\delta](f\ w^1, f\ w^2)$. We may write δ in the form $\delta = \{(h_1, l_1), \ldots, (h_n, l_n)\}$. If $w_1 \neq \bot$ we have

$$\begin{aligned}
&\mathrm{FH}(f\ w^1) = \mathrm{FH}(w_0 \texttt{++} w^1 \texttt{++} w_1 \texttt{++} \ldots \texttt{++} w^1 \texttt{++} w_n) = \mathrm{FH}(w^1)\\
&(f\ w^1)[l_1/h_1, \ldots, l_n/h_n]\\
&\qquad = (w_0 \texttt{++} w^1 \texttt{++} w_1 \texttt{++} \ldots \texttt{++} w^1 \texttt{++} w_n)[l_1/h_1, \ldots, l_n/h_n]\\
&\qquad = f(w^1[l_1/h_1, \ldots, l_n/h_n]
\end{aligned}$$

and then the desired result follows from the assumptions. If $w^1 = \bot$ we also have $w^2 = \bot$ so that $f\ w^1 = f\ w^2$. The result is then immediate if $f\ w^1 = f\ w^2 = \bot$ so assume that $f\ w^1 = f\ w^2 \neq \bot$. Then $f\ w^1 = f\ w^2 = f\ [\,]$ since $w^1 \sqsubseteq [\,]$ implies $f\ w^1 \sqsubseteq f\ [\,]$ and by flatness of CHARLIST equality follows. Next

$$\begin{aligned}
&\mathrm{FH}(f\ [\,]) = \mathrm{FH}(w_0 \texttt{++} w_1 \texttt{++} \ldots \texttt{++} w_n) = \emptyset\\
&(f\ [\,])[l_1/h_1, \ldots, l_n/h_n] = f\ [\,]
\end{aligned}$$

and the result follows. □

In the examples we have given in this section the substitution property is something that may or may not be satisfied by the functions defined: it holds for `letter` but not for `badge`. In the applications to code generation [9, 10] that motivated the present development, the situation is slightly different. There the substitution property does hold for all functions defined (in the process of interpreting a metalanguage). The need to formulate the substitution property then arises because of the higher-order constant `fix`. It will be instructive to regard `fix` as having functionality

$$(\texttt{charlist} \to \texttt{charlist}) \to \texttt{charlist}$$

and to regard elements of `charlist` as pieces of code. Any function (of functionality `charlist` $\rightarrow$ `charlist`) that is passed as a parameter to `fix` will satisfy the substitution property because of the way it is defined. However, the domain for `charlist` $\rightarrow$ `charlist` contains many functions that do not and hence the definition of `fix` cannot make that assumption without formally defining a predicate like $SUBST$ that expresses the substitution property.

This phenomenon has some connection to the question of full abstractness. Usually full abstractness means that equality of denotations is equivalent to equal behaviour in all program contexts. For our purposes it is more instructive to use the characterization of [3][1]: full abstractness holds provided all (compact) elements of the domains are denotable. The search for full abstractness then may be regarded as a search for models where there are no superfluous elements. Usually the superfluous elements express some features corresponding to parallel evaluation of arguments with the ability to retract a (possibly nonterminating) evaluation. For us the superfluous elements of `charlist` $\rightarrow$ `charlist` are those that do not satisfy the substitution property. The fact that no fully abstract models are known to exist for simple sequential languages suggests that techniques like our $SUBST$ relation may be unavoidable.

Historical Remark The concept of Kripke-logical relations, including the result on structural induction (our Lemma 2.6) was already studied in [12]. Further studies along these lines may be found in [5] and a brief survey is contained in [4]. Independently of the latter works the authors used the concept in [9]. The use of Kripke-logical relations to achieve substitution properties (our Lemma 2.13) is due to [9] with preliminary ideas in [7]. (The main limitation of [7] is that all sets $\{(h_1, l_1), \ldots, (h_n, l_n)\}$ have $m = 1$ so that the development only allows the fixed point operator to occur once.)

3 Kripke-Layered Predicates

We now return to the challenge of producing well-structured proofs and for this to succeed our techniques must interact well with the concepts of log-

[1]The central result of [3] is the "Second Context Lemma".

ical and Kripke-logical relations. In Example 1.3 we managed to conduct a proof about an aggregate concept by conducting proofs about each constituent concept and then combining the results. However, we did indicate that this approach would not work in general and we shall give an example shortly. Furthermore we shall present an example showing that in general the constituent concepts must be allowed to depend on each other.

Example 3.1 As an extension of Examples 1.1 and 1.3 consider the following module:

$$\begin{array}{l}\texttt{spec comp1: } (\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num}) \\ \texttt{with comp1 g maps nonnegative integers to nonnegative} \\ \qquad \texttt{integers, provided that g does} \\ \texttt{impl comp1} = \lambda_{\texttt{g}}.\ \lambda \texttt{x}.\ \frac{\texttt{g(x)} * \texttt{g(x)}}{2} + \frac{\texttt{g(x)}}{2}\end{array}$$

Clearly `comp1` satisfies its interface condition because `comp1 g` $=$ `sum1` $\circ$ `g` and we happen to know that `comp1` $\models (NONNEG \wedge INT)[\texttt{num} \to \texttt{num}]$ and we assume that `g` $\models (NONNEG \wedge INT)[\texttt{num} \to \texttt{num}]$ and so may use the proof rule

$$\frac{R_{t_1 \to t_2}(f_1)\ R_{t_2 \to t_3}(f_2)}{R_{t_1 \to t_3}(f_2 \circ f_1)}$$

which holds for any logical relation R, in particular $NONNEG \wedge INT$. However, this was not a simple proof by structural induction because it included algebraic transformations on `comp1 g`. To achieve a proof more along the lines of Example 1.3 we may begin by establishing that

$$\begin{array}{l}\texttt{comp1} \models NONNEG[(\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num})] \\ \texttt{comp1} \models INT[(\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num})]\end{array}$$

using the methods sketched in Example 1.3. Next we may aim at showing

$$\texttt{sum1} \models (NONNEG \wedge INT)[(\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num})]$$

using the proof rule

$$\frac{e \models R'[t] \quad e \models R''[t]}{e \models (R' \wedge R'')[t]} \quad \text{if } t \ldots$$

in the instance where $t = (\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num})$, $R' = NONNEG$, and $R'' = INT$.

Validation of this instance fails, intuitively because it relies on the validity of the rule instances

$$\frac{e \models (NONNEG \wedge INT)[\texttt{num} \to \texttt{num}]}{e \models NONNEG[\texttt{num} \to \texttt{num}]}$$

$$\frac{e \models (NONNEG \wedge INT)[\texttt{num} \to \texttt{num}]}{e \models INT[\texttt{num} \to \texttt{num}]}$$

$$\frac{e \models (NONNEG \wedge INT)[\texttt{num} \to \texttt{num}] \; e \models INT[\texttt{num} \to \texttt{num}]}{e \models NONNEG \wedge INT[\texttt{num} \to \texttt{num}]}$$

The latter instance is valid (as seen in Example 1.3). For the first two instances note that taking e to be

$$\begin{array}{ll} \lambda x. & \text{if } x = -1 \text{ then } \frac{1}{2} \text{ else} \\ & \text{if } x = \frac{1}{2} \text{ then } -1 \text{ else} \\ & x \end{array}$$

invalidates both: the above function maps nonnegative integers to nonnegative integers but does not map nonnegative numbers to nonnegative numbers nor does it map integers to integers. □

Example 3.2 The previous example showed that a function might preserve an aggregation of properties but none of the constituent properties individually. Perhaps more "natural" are the settings where only one of the constituent properties is preserved individually. For an example of this consider the following slight variation on the module of Examples 1.1 and 1.3:

```
spec  sum0: (num → num)
with  sum0 g maps nonnegative integers to
      nonnegative integers,
      /* sum0 x = 0 + ... (x-1) */
impl  sum0 = λg. λx. x*x/2 + x/2
```

Proceeding along the lines of Example 1.3 we might aim at first proving

$$\texttt{sum0} \models \mathit{NONNEG}[\texttt{num} \to \texttt{num}]$$

but this fails: `sum0` maps $\frac{1}{3}$ to $-\frac{1}{9}$. On the other hand

$$\texttt{sum0} \models \mathit{INT}[\texttt{num} \to \texttt{num}]$$

succeeds. We can remedy our failure by showing that `sum0` maps nonnegative integers to nonnegative numbers; this amounts to strengthening our assumptions without also strengthening our proof obligation. We shall allow to write this succinctly as

$$\texttt{sum0} \models (\mathit{INT} \wedge \mathit{NONNEG})[\texttt{num}] \Rightarrow \mathit{NONNEG}[\texttt{num}]$$

To prove the desired

$$\texttt{sum0} \models (\mathit{INT} \wedge \mathit{NONNEG})[\texttt{num} \to \texttt{num}]$$

we may then attempt to use the proof rule

$$\frac{e \models R'[t_1 \to t_2] \quad e \models (R' \wedge R'')[t_1] \Rightarrow R''[t_2]}{e \models (R' \wedge R'')[t_1 \to t_2]} \text{ if } t_1, t_2 \ldots$$

in the instance where $t_1 = t_2 = \texttt{num}$, $R' = \mathit{INT}$, and $R'' = \mathit{NONNEG}$.

Validation of this instance succeeds because we have the following valid rule instances

$$\frac{e \models (\mathit{INT} \wedge \mathit{NONNEG})[\texttt{num}]}{e \models \mathit{INT}[\texttt{num}]}$$

$$\frac{e \models \mathit{INT}[\texttt{num}] \quad e \models \mathit{NONNEG}[\texttt{num}]}{e \models (\mathit{INT} \wedge \mathit{NONNEG})[\texttt{num}]}$$

and hence our proof is complete.

However, in general the above proof rule fails when t_1 or t_2 is a function type, intuitively because the analogues of the two rule instances above then may fail. □

These examples show that our problems are due to lack of introduction rules for proving $(R' \wedge R'')_t(w)$ given $R'_t(w)$ and $R''_t(w)$, and a lack of elimination rules for proving $R'_t(w)$ and $R''_t(w)$ given $(R' \wedge R'')_t(w)$. The latter example additionally suggests that it may be more appropriate to consider R'' as a way to extend R' rather than a predicate that should be preserved on its own; equivalently, to establish R'' of a result we need to know R' in addition to R'' of the argument. Our solution therefore will be to define a new notion of a layered combination of R' and R'' such that useful introduction and elimination rules do become valid.

Recall that a logical relation may be regarded as a Kripke-logical relation over a partially ordered set with just one element. For conciseness we therefore concentrate on the more general case.

Definition 3.3 Given Kripke-indexed relations P and Q over Δ and $\mathcal{I}_1, \ldots, \mathcal{I}_m$, we define a Kripke-indexed relation $P \,\&\, Q$ over Δ and $\mathcal{I}_1, \ldots, \mathcal{I}_m$, as follows:

$$
\begin{array}{lcl}
(P \,\&\, Q)_{\mathtt{num}}[\delta](w_1, \ldots, w_m) & \equiv & P_{\mathtt{num}}[\delta](w_1, \ldots, w_m) \wedge Q_{\mathtt{num}}[\delta](w_1, \ldots, w_m) \\
(P \,\&\, Q)_{\mathtt{bool}}[\delta](w_1, \ldots, w_m) & \equiv & P_{\mathtt{bool}}[\delta](w_1, \ldots, w_m) \wedge Q_{\mathtt{bool}}[\delta](w_1, \ldots, w_m) \\
(P \,\&\, Q)_{\mathtt{charlist}}[\delta](w_1, \ldots, w_m) & \equiv & P_{\mathtt{charlist}}[\delta](w_1, \ldots, w_m) \wedge \\
 & & \qquad Q_{\mathtt{charlist}}[\delta](w_1, \ldots, w_m) \\
(P \,\&\, Q)_{t_1 \to t_2}[\delta](f_1, \ldots, f_m) & \equiv & P_{t_1 \to t_2}[\delta](f_1, \ldots, f_m) \wedge \\
 & & \forall \delta' \sqsupseteq \delta : \forall (w_1, \ldots, w_n) : \\
 & & \qquad (P \,\&\, Q)_{t_1}[\delta'](w_1, \ldots, w_m) \\
 & & \qquad \Downarrow \\
 & & \qquad (P \,\&\, Q)_{t_2}[\delta'](f_1\ w_1, \ldots, f_m\ w_m)
\end{array}
$$

We shall say that $P \,\&\, Q$ is a Kripke-layered predicate over Δ and $\mathcal{I}_1, \ldots, \mathcal{I}_m$ and that it is the Kripke-layered combination of P and Q.

When P and Q are indexed relations (in the sense of Section 1) we shall say that $P \,\&\, Q$ is a layered predicate and that it is the layered combination of

P and Q. □

Fact 3.4 With P and Q as above, $P \,\&\, Q$ is a Kripke-indexed relation over Δ and $\mathcal{I}_1, \ldots, \mathcal{I}_m$. However, $P \,\&\, Q$ need not be Kripke-logical even when P and Q both are. □

Proof. Only the latter claim is nontrivial. It suffices to find an instance where

$$\begin{array}{lll} \forall \delta' \sqsupseteq \delta : \forall (w_1, \ldots, w_m) : & & (P \,\&\, Q)_{t_1}[\delta'](w_1, \ldots, w_m) \\ & \Downarrow & \\ & & (P \,\&\, Q)_{t_2}[\delta'](f_1\ w_1, \ldots, f_m\ w_m) \end{array}$$

does not imply $P_{t_1 \to t_2}[\delta](f_1, \ldots, f_m)$. But this was established by the final part of Example 3.1. □

Fact 3.5 With P and Q as above

$$(P \,\&\, Q)_t[\delta](w_1, \ldots, w_m) \Rightarrow P_t[\delta](w_1, \ldots, w_m)$$

holds for all types t. □

Proof. This is a structural induction over t. In all cases the result follows because P_t is an explicit conjunct in $(P \,\&\, Q)_t$. □

Despite the negative statement in Fact 3.4, the concept of Kripke-layered predicates interacts well with the semantics of the λ-calculus. Recalling Definition 2.5 we have:

Lemma 3.6 *The Kripke-Zayered predicate $P\&Q$ admits structural induction provided P does.* □

Proof. Using the notation of Definition 2.5, this is a structural induction over e. For a variable v the result is immediate from the assumptions. This is also the case for a constant c. For an application $e_1\ e_2$ we use the induction hypothesis to obtain

$$\begin{array}{l} (P\&Q)_{t' \to t}[\delta](\mathcal{I}_1[\![e_1]\!]_{tenv}(w_1^1, \ldots, w_n^1), \ldots, \mathcal{I}_m[\![e_1]\!]_{tenv}(w_1^m, \ldots, w_n^m)) \\ (P \,\&\, Q)_{t'}[\delta](\mathcal{I}_1[\![e_2]\!]_{tenv}(w_1^1, \ldots, w_n^1), \ldots, \mathcal{I}_m[\![e_2]\!]_{tenv}(w_1^m, \ldots, w_n^m)) \end{array}$$

and then we use the definition of $P \,\&\, Q$; to be more specific we use the following proof rule

$$\frac{(P \,\&\, Q)_{t' \to t}[\delta](f_1, \ldots, f_m)(P \,\&\, Q)_{t'}[\delta](w_1, \ldots, w_m)}{(P \,\&\, Q)_t[\delta](f_1(w_1), \ldots, f_m(w_m))}$$

that follows immediately from the definition of $P \,\&\, Q$. For a λ-abstraction $\lambda v : t'.e$ we must show two results. One is that

$$P_{t' \to t}[\delta](\mathcal{I}_1[\![\lambda v : t'.e]\!]_{tenv}(w_1^1, \ldots, w_n^1), \ldots, \mathcal{I}_m[\![\lambda v : t'.e]\!]_{tenv}(w_1^m, \ldots, w_n^m))$$

but this follows from the assumptions, Fact 3.5 and that P admits structural induction. The other amounts to choosing $\delta' \sqsupseteq \delta$ and assume that

$$(P \,\&\, Q)_{t'}[\delta'](w_{n+1}^1, \ldots, w_{n+1}^m)$$

and then show

$$(P \,\&\, Q)_t[\delta'](\mathcal{I}_1[\![\lambda v : t'.e]\!]_{tenv}(w_1^1, \ldots, w_n^1)(w_{n+1}^1), \ldots, \\ \mathcal{I}_m[\![\lambda v : t'.e]\!]_{tenv}(w_1^m, \ldots, w_n^m)(w_{n+1}^m))$$

But by assumption we have

$$(P \,\&\, Q)_{t_i}[\delta](w_i^1, \ldots, w_i^m) \quad \text{for } i = 1, \ldots, n$$

so using the proof rule

$$\frac{(P \,\&\, Q)_t[\delta](w_1, \ldots, w_m)}{(P \,\&\, Q)_t[\delta'](w_1, \ldots, w_m)} \; \delta' \sqsupseteq \delta$$

we obtain

$$(P \,\&\, Q)_{t_i}[\delta'](w_i^1, \ldots, w_i^m) \text{ for } i = n + 1$$

It follows from the induction hypothesis that

$$(P \,\&\, Q)_t[\delta'](\mathcal{I}_1[\![e]\!]_{tenv,(v,t')}(w_1^1,\ldots,w_n^1,w_{n+1}^1),\ldots,\\ \mathcal{I}_m[\![e]\!]_{tenv,(v,t')}(w_1^m,\ldots,w_n^m,w_{n+1}^m))$$

and since

$$\mathcal{I}_i[\![\lambda v:t'.e]\!]_{tenv}(w_1^i,\ldots,w_n^i)(w_{n+1}^i) = \mathcal{I}_i[\![e]\!]_{tenv,(v,t')}(w_1^i,\ldots,w_n^i,w_{n+1}^i)$$

this is the desired result. □

Corollary 3.7 If P is Kripke-logical, then $P \& Q$ admits structral induction.

□

Taking $n = 0$ and $m = 1$ we get:

Corollary 3.8 If the closed expression e has type t, i.e. $(\ ) \vdash e : t$, and $c' \models (P \,\&\, Q)[t']$ for all constants c' of type t' occurring in e, then $e \models (P \,\&\, Q)[t]$ provided that $P \,\&\, Q$ is the Kripke-layered combination of Kripke-logical relations P and Q. □

To complement Lemma 3.6 we also need to consider how to prove that $P \,\&\, Q$ holds for the constants. In the previous sections there was little to say because there the predicates had "no structure", but here the predicate are combinations of other predicates. This amounts to the study of introduction rules for $P \,\&\, Q$ and for the sake of completeness we shall give elimination rules and a few derived rules as well. We begin with the elimination rules:

$$\text{[E1]}\quad \frac{(P \,\&\, Q)_t[\delta](w_1,\ldots,w_m)}{P_t[\delta](w_1,\ldots,w_m)}$$

$$\text{[E2]}\quad \frac{(P \,\&\, Q)_t[\delta](w_1,\ldots,w_m)}{Q_t[\delta](w_1,\ldots,w_m)} \quad \text{if } t \text{ is a base type}$$

The validity of the first rule, for arbitrary t and δ, is a direct consequence of Fact 3.5. The validity of the second rule, for $t \in \{\texttt{num, bool, charlist}\}$, is a direct consequence of the definition of $P \,\&\, Q$; that the rule may fail for function types follows rather easily from Example 3.2. A derived rule that occasionally is useful is

$$(P \,\&\, Q)_{t_n \to \ldots \to t_0}[\delta](f)$$
$$P_{t_n}[\delta](w_n)$$
$$\vdots$$

$$[\mathrm{E1}'] \qquad \frac{P_{t_1}[\delta](w_1)}{P_{t_0}[\delta](f\ w_n \ldots w_1)} \qquad \text{if } P \text{ is Kripke-logical}$$

(where for simplicity we took $m = 1$). Turning to the introduction rules it is helpful to say that t is an *iterated base type* (of order n) whenever $\exists\, t_n, \ldots, t_1, t_0 \in \{\mathtt{num}, \mathtt{bool}, \mathtt{charlist}\} : t = t_n \to t_{n-1} \to \ldots \to t_1 \to t_0$.

We then have

$$[\mathrm{I}] \quad \frac{P_t[\delta](w_1, \ldots, w_m) \quad Q_t[\delta](w_1, \ldots, w_m)}{(P \,\&\, Q)_t[\delta](w_1, \ldots, w_m)}$$
if t is an iterated base type and P and Q are Kripke-logical.

Fact 3.9 The above rule is valid. □

Proof. Validity is proven by induction on the order n of the iterated base type t. The base case is trivial given the definition of $P \,\&\, Q$ on base types. For the inductive step we consider $t = t_{n+1} \to (t_n \to \ldots \to t_0)$ and assume

$$P_t[\delta](f_1, \ldots, f_m)$$
$$Q_t[\delta](f_1, \ldots, f_m)$$

and must show $(P \,\&\, Q)_t[\delta](f_1, \ldots, f_m)$. This amounts to showing

$$P_t[\delta](f_1, \ldots, f_m)$$

which is trivial given the assumptions, and to consider $\delta' \sqsupseteq \delta$ and assume

$$(P \,\&\, Q)_{t_{n+1}}[\delta'](w_1, \ldots, w_m)$$

and show $(P \,\&\, Q)_{t_n \to \ldots \to t_0}[\delta'](f_1\ w_1, \ldots, f_m\ w_m)$. But from our two elimination rules we have $P_{t_{n+1}}[\delta'](w_1, \ldots, w_m)$ and $Q_{t_{n+1}}[\delta'](w_1, \ldots, w_m)$ so that our assumptions yield

$$P_{t_n\to\ldots\to t_0}[\delta'](f_1\ w_1,\ldots,f_m\ w_m)$$
$$Q_{t_n\to\ldots\to t_0}[\delta'](f_1\ w_1,\ldots,f_m\ w_m)$$

The desired result then follows from the induction hypothesis. □

It is possible to generalize this rule in various ways. One rule that my be useful is

$$[I'] \quad \frac{P_{t_n\to\ldots\to t_0}[\delta](f) \qquad (\forall i\in\{1,\ldots,n\}:(P\ \&\ Q)_{t_i}[\delta](w_i))\Rightarrow Q_{t_0}[\delta](f\ w_n\ldots w_1)}{(P\ \&\ Q)_{t_n\to\ldots\to t_0}[\delta](f)}$$

if t_0 is an iterated base type and P and Q are Kripke-logical

(where for simplicity we took $m = 1$); alternatively one could use a Gentzen-style presentation. Validity of this rule may be shown by numerical induction on n.

Example 3.10 We now briefly return to the successes and failures encountered in Examples 1.3, 3.1 and 3.2. In all three examples the predicate of interest is

$$INT\ \&\ NONNEG$$

rather than $NONNEG \wedge INT$. Concerning Example 1.3 we were able to show

$$\texttt{sum1} \models INT[\texttt{num}\to\texttt{num}]$$
$$\texttt{sum1} \models NONNEG[\texttt{num}\to\texttt{num}]$$

and the desired

$$\texttt{sum1} \models (INT\ \&\ NONNEG)[\texttt{num}\to\texttt{num}]$$

then is a simple application of Introduction Rule $[I]$, since $\texttt{num}\to\texttt{num}$ is an iterated base type. This should hardly be surprising since this is the same approach that succeeded in Example 1.3 but for $NONNEG \wedge INT$, i.e. $INT \wedge NONNEG$, instead of $INT\ \&\ NONNEG$.

Concerning Example 3.1 we noted that the obvious approach is to begin by showing

$$\begin{aligned}&\texttt{comp1} \models \mathit{INT}[\texttt{num} \to \texttt{num}]\\ &\texttt{comp1} \models \mathit{NONNEG}[\texttt{num} \to \texttt{num}]\end{aligned}$$

and this still succeeds. However, we still cannot achieve the desired result because Introduction Rule $[I]$ is not applicable as $(\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num})$ is not an iterated base type. Instead we aim at using the stronger rule $[I']$. For this we modify the second claim above to

$$\begin{aligned}\texttt{comp1} \models &(\mathit{INT}\ \&\ \mathit{NONNEG})[\texttt{num} \to \texttt{num}]\\ &\Rightarrow (\mathit{INT}\ \&\ \mathit{NONNEG})[\texttt{num}]\\ &\Rightarrow \mathit{NONNEG}[\texttt{num}]\end{aligned}$$

where we have used " ... $\Rightarrow$ ... " in the sense explained in Example 3.2. This succeeds and we may then used rule $[I']$ to obtain the desired result.

Finally Example 3.2 goes through in much the same way as above; first note that

$$\begin{aligned}&\texttt{sum0} \models \mathit{INT}[\texttt{num} \to \texttt{num}]\\ &\texttt{som0} \models (\mathit{INT}\ \&\ \mathit{NONNEG})[\texttt{num}] \to \mathit{NONNEG}[\texttt{num}]\end{aligned}$$

and then use rule $[I']$ to obtain the desired result. □

Example 3.11 A "realistic example" is beyond the space available to us but we can give a very sketchy overview of the development of [10, Chapter 6]. The problem under study is that of translating a certain typed λ-calculus into an abstract mashine somewhat similar to the categorical abstract machine. The λ-calculus allows arbitrary nesting of fixed point operators and this is the root of our first problem. The abstract machine works in a stack-like manner and this allows to separate the correctness proof into two parts. To be more specific the correctness predicate is of the form $(\mathcal{R}_1\ \&\ \mathcal{R}_2)\ \&\ \mathcal{R}_3$.

The first predicate, $\mathcal{R}_1$, aims at establishing a substitution property along the lines of the Extended Example of Section 2; thus $\mathcal{R}_1(w)$ roughly means

$SUBST[\emptyset](w, w)$. This is necessary for the semantics of the abstract machine to behave as desired. This is because the code for the fixed point of a functional is the code resulting from supplying the functional with an appropriate call instruction that "points" to that code. The abstract machine then executes a call instruction by replacing it with yet another copy of that code. Having done this once the unfolded code, viewed in its original context, should desirably correspond to the result of applying the functional to that code. In symbols this amounts to

$$(F\ \texttt{call})[F\ \texttt{call}/\ \texttt{call}] = F(F\ \texttt{call})$$

where the functional is written as F. To achieve this we use the substitution property and we refer to [10, Section 6.2] for the details.

The second predicate, $\mathcal{R}_2$, aims at showing that the code generated from well-formed λ-expressions is well-behaved. Certainly the execution of a piece of code can result in errors as well as nontermination. However, there are several "structural properties" that will be fulfilled by the code generated although they may not hold for arbitrary code sequences. The typical example of this is that evaluation of an expression on a stack pushes an element upon the stack and leaves the remainder of the stack unchanged. In [10] the λ-calculus and the code generation is such that instead evaluation of an expression only modifies the top element of the stack and leaves the remaining elements and the height of the stack unchanged. There are several complications in the definition of the predicate because the elements on the stack may themselves contain code components. We refer to [10, Section 6.3] for the detailed definition and proofs.

The third predicate, $\mathcal{R}_3$, then finally expresses the correctness of the code generated with respect to the semantics of the λ-expression. The detailed development rather closely follows that of well-behavedness except that at each step the correctness considerations need to be added. We refer to [10, Section 6.4] for the details.

Overall the development of [10, Section 6] is almost 70 pages with about 50 pages devoted to establishing $(\mathcal{R}_1 \,\&\, \mathcal{R}_2) \,\&\, \mathcal{R}_3$. We strongly believe that the "separation of concerns" facilitated by expressing the desired predicate as a combination of three simpler predicates, and by proving the desired predicate in three stages, is of immense help when developing the proof as well as when

presenting it to others. □

Generalizations

It follows from Corollary 3.7 and Fact 3.4 that the Kripke-logical relations constitute a proper subset of those Kripke-indexed relations that admit structural induction. This suggests studying a notion of "benign" modifications of Kripke-logical relations so as to obtain a larger subset.

We have no formal definition of "benign" but the general idea is that the definition of the Kripke-indexed predicate P (over Δ and $\mathcal{I}_1, \ldots, \mathcal{I}_m$) is given by a formula

$$\begin{array}{l} P_t[\delta](w_1, \ldots, w_m) \equiv \forall \bar{\delta} \in \bar{\Delta} : \forall \bar{w} \in D_t : \\ \quad P'_t[\partial(\delta, \bar{\delta})](\omega_1^t(w_1, \ldots, w_m, \bar{w}), \ldots, \omega_{m'}^t(w_1, \ldots, w_m, \bar{w})) \end{array}$$

where

$$\begin{array}{l} \partial : \Delta \times \bar{\Delta} \to \Delta' \\ \omega_i^t : \mathcal{I}_1[\![t]\!] \times \ldots \times \mathcal{I}_m[\![t]\!] \times D_t \to \mathcal{I}'_i[\![t]\!] \end{array}$$

and $\bar{\Delta}$ is a non-empty partially ordered set, D_t a (non-empty) domain that depends on t and P' is a Kripke-logical relation over Δ' and $\mathcal{I}'_1, \ldots, \mathcal{I}'_m$.

To simplify matters let us make the rather drastic assumption that each ω_i^t selects one of its first m arguments, i.e. $\omega_i^t(w_1, \ldots, w_m, \bar{w}) = w_{n_i}$ for $n_i \in \{1, \ldots, m\}$, and that the corresponding interpretations agree, i.e. $\mathcal{I}'_i = \mathcal{I}_{n_i}$. Then P admits structural induction. To see this, note that the assumptions

$$\begin{array}{l} P_{t_i}[\delta](w_i^1, \ldots, w_i^m) \text{ for } \ldots \\ P_{t'}[\delta](\mathcal{I}_1(c'), \ldots, \mathcal{I}_m(c')) \text{ for } \ldots \end{array}$$

amount to

$$\begin{array}{l} P'_{t_i}[\partial(\delta, \bar{\delta})](\omega_i^{t_i}(w_i^1, \ldots, w_i^m), \ldots) \text{ for } \ldots \\ P'_{t'}[\partial(\delta, \bar{\delta})](\omega_i^{t'}(\mathcal{I}_1(c'), \ldots, \mathcal{I}_m(c')), \ldots) \text{ for } \ldots \end{array}$$

for all choices of $\bar{\delta} \in \bar{\Delta}$ and where we have dropped the $\bar{w}$ argument. For each $\bar{\delta} \in \bar{\Delta}$, Lemma 2.6 (and Definition 2.5) then gives

$$P'_t[\partial(\delta, \bar{\delta})](\mathcal{I}'_1[\![e]\!]_{tenv}(\omega_1^{t_1}(w_1^1, \ldots, w_1^m), \ldots, \omega_1^{t_n}(w_n^1, \ldots, w_n^m)), \ldots)$$

and this amounts to

$$P'_t[\partial(\delta, \bar{\delta})](\omega_1^t(\mathcal{I}_1[\![e]\!]_{tenv}(w_1^1, \ldots, w_n^1), \ldots, \mathcal{I}_m[\![e]\!]_{tenv}(w_1^m, \ldots, w_n^m)), \ldots)$$

from which

$$P_t[\delta](\mathcal{I}_1[\![e]\!]_{tenv}(w_1^1, \ldots, w_n^1), \ldots, \mathcal{I}_m[\![e]\!]_{tenv}(w_1^m, \ldots, w_n^m))$$

follows.

We already used a result along these lines in Example 3.11 (and [10]): while *SUBST* is a Kripke-logical relation, the relation $\mathcal{R}_1$ is not although it is a "benign" modification of *SUBST*. Thus $\mathcal{R}_1$ does admit structural induction and by Lemma 3.6 so do $\mathcal{R}_1 \,\&\, \mathcal{R}_2$ and $(\mathcal{R}_1 \,\&\, \mathcal{R}_2) \,\&\, \mathcal{R}_3$.

In another direction we may generalize the number of P's and Q's considered. Specifically one may define

$$(P_1, \ldots, P_p) \,\&\, (Q_1, \ldots, Q_q)$$

for $p \geq 1$ and $q \geq 1$. For a base type to $t_0 \in \{\texttt{num}, \texttt{bool}, \texttt{charlist}\}$ we set

$$\begin{array}{l}((P_1, \ldots, P_p) \,\&\, (Q_1, \ldots, Q_q))_{t_0}[\delta](w_1, \ldots, w_m) \equiv \\ \quad \bigwedge_{i=1}^{p}(P_i)_{t_0}[\delta](w_1, \ldots, w_m) \wedge \bigwedge_{j=1}^{q}(Q_j)_{t_0}[\delta](w_1, \ldots, w_m)\end{array}$$

and for a function type $t_1 \to t_2$ we set

$$\begin{array}{l}((P_1, \ldots, P_p) \,\&\, (Q_1, \ldots, Q_q))_{t_1 \to t_2}[\delta](f_1, \ldots, f_m) \equiv \\ \quad \bigwedge_{i=1}^{p}(P_i)_{t_1 \to t_2}[\delta](f_1, \ldots, f_m) \wedge \forall \delta' \sqsupseteq \delta : \forall (w_1, \ldots, w_m) : \\ \qquad\qquad ((P_1, \ldots, P_p) \,\&\, (Q_1, \ldots, Q_q))_{t_1}[\delta](w_1, \ldots, w_m) \\ \qquad \Downarrow \\ \qquad\qquad ((P_1, \ldots, P_p) \,\&\, (Q_1, \ldots, Q_q))_{t_2}[\delta](f_1\ w_1, \ldots, f_m\ w_m)\end{array}$$

Taking $p > 1$ defines a more general notion that may well be useful; taking $q > 1$ is useless as $(P_1, \ldots, P_p) \& (Q_1, \ldots, Q_q)$ is equivalent to $(P_1, \ldots, P_p) \& (Q_1 \wedge \ldots \wedge Q_q)$ where $(Q_1 \wedge \ldots \wedge Q_q)$ is defined as in Example 1.3.

Historical Remark The notion of (Kripke-) layered predicate is based on [10, Chapter 6] which is the only relevant reference that we know of.

4 Conclusion

We have presented a number of techniques for the defining predicates so as to allow proofs by structural induction. All of these are based on the underlying concept of logical relations and have been applied to problems with substance; we refer to [10] and its bibliography for examples. Some of the main lessons learned may be summarized as follows:

- Some predicates have base cases that are most naturally expressed at level 1 (functions between base values) rather than at level 0 (base values). To adopt logical relations the notion of *partial equivalence relations* is useful or more generally using the same interpretation more than once. (Counting the levels from one rather than zero this also explains the distinction between "first-order" and "second-order" made in [6].)
- *Kripke-layered relations* have "local memory" consisting of the parameter (δ) drawn from the partially ordered set (Δ). This allows them to be used to describe a substitution property by means of a level 0 behavior.
- *Kripke-layered predicates* are not simply Kripke-logical relations (Fact 3.4) but allow for more structured proofs that proceed in stages. The complexity of each stage is significantly smaller (but often still substantial) than a brute force proof.

The strengths of Kripke-layered predicates include the ability to reorder the parameters and to modify the partially ordered set over which the parameters

are drawn. Comparing the proof of [10, Chapter 6] with that of [9], where only Kripke-logical relations were used, we believe that the advantages claimed for Kripke-layered predicates are indeed sustained.[2]

The notions studied in this paper are fairly robust. One may add additional type constructors like sum, product and recursive types and still perform the development. Also one may restrict the attention to admissible predicates so as to support Scott-induction and the development still carries through. Finally, when no recursive types or fixed point constructs are present one may use ordinary sets instead of domains. This all calls for a more general categorical formulation of Kripke-layered predicates and this should also include a more general theory of "benign" modification.

Acknowledgement

This work was supported in part by The Danish Research Councils under grant 5.21.08.03 ("The DART-Project"). Torben Amtoft and Torben Lange provided useful comments and Karen Møller expert typing.

References

[1] S. Hunt: PERs Generalise Projections for Strictness Analysis, report DOC 90/14, Imperial College (1990).

[2] S Hunt, D. Sands: Binding Time Analysis: A New PERspective, *Proc. ACM Symposium on Pascal Evaluation and Semantics-Based Program Manipulation,* ACM Press (1991) 154-165.

[3] R. Milner: Fully abstract models of typed λ-calculi, *Theoretical Computer Science* **4** (1977) 1-22.

[4] J.C. Mitchell: Type Systems for Programming Languages, in: *Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics,* J. van Leeuwen (ed.), Elsevier (1990).

[2]The relative success of [9] also raises the question whether Kripke-layered predicates are more intimately connected to Kripke-logical relations than suggested by Fact 3.4.

[5] J.C. Mitchell, E. Moggi: Kripke-style models for typed λ-calculus, *Proc. 2nd Ann. IEEE Symposium on Logic in Computer Science,* IEEE Press (1987) 303-314.

[6] F. Nielson: Program Transformations in a Denotational Setting, *ACM Transactions on Programming Languages and Systems* **7** (1985) 359-379.

[7] F. Nielson: Correctness of code generation from a two-level metalanguage, *Proc. ESOP 1986, Springer Lecture Notes in Computer Science* **213** (1986) 30-40.

[8] F. Nielson: Strictness Analysis and Denotational Abstract Interpretation, *Information and Computation* **76** 29-92.

[9] F. Nielson, H.R. Nielson: Two-Level Semantics and Code Generation, *Theoretical Computer Science* **56** (1988) 59-133.

[10] F. Nielson, H.R. Nielson: *Two-Level Functional Languages, Cambridge Tracts in Theoretical Computer Science* **34**, Cambridge University Press (1992)

[11] G.D. Plotkin: Lambda-definability and logical relations, Edinburgh AI memo, Edinburgh University (1973).

[12] G.D. Plotkin: Lambda-definability in the fill Type Hierarchy, in: To H.B. Curry: *Essays on Combinatory Logic, Lambda Calculus, and Formalism,* J.P. Seldin and J.R. Hindley (eds.), Academic Press (1980).

[13] J.C. Reynolds: On the relation between direct and continuation semantics, *Proc. 2nd ICALP, Springer Lecture Notes in Computer Science* **14** (1974).