Deterministic Expressions in C

Michael Norrish

Computer Laboratory
University of Cambridge**
mn200@cl.cam.ac.uk

Abstract. Expressions in the programming language C have such an under-specified semantics that one might expect them to be non-deterministic. However, with the help of a mechanised formalisation, we have shown that the semantics' additional constraints actually result in a large class of C expressions having only one possible behaviour.

1 Introduction

The semantics of the programming language C is specified in an ISO standard [3]. However, this semantics is written in natural language, and is thus unsuitable as the basis for formal work such as verification. Indeed, there are a number of unresolved disputes about various details in this standard.[1]

However, our Cholera formalisation [6,7] is a completely formal semantics for the bulk of the C language. It is formulated in a structural operational style (see, for example, [2]) and is embedded in the HOL theorem prover [1]. On this basis, it *is* possible to prove facts about the C language (modulo the degree of certainty with which one believes the formalisation to be correct). For example, it is possible to derive various "axiomatic" rules that allow one to reason about C programs with Hoare-like triples, as described in [5,7].

The work described here considers the semantics of C expressions, and in particular demonstrates that a significant class of these expressions are deterministic. This is an important result in the context of verification because it allows one to perform a verification with respect to just one possible path of execution. Otherwise, if an expression can evaluate in n different ways, then any verification of a program that contains it must demonstrate that the final post-condition holds for all n possibilities, a tedious task at best.

** Fax: +44 1223 334678

[1] See for example the Usenet newsgroup `comp.std.c`, where issues such as whether or not function calls may interleave are debated.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 147–161, 1999.

In addition to defining the semantics of C, our Cholera project aims to put results like this determinism theorem to work: using them in the verification of not entirely trivial programming examples. Such verification examples will support the thesis that verification of programs in complicated programming languages is possible, particularly if one has mechanical support for the task. Moreover, the fact that proofs of this nature are practical is an indication that even programming language semantics can be satisfactorily mechanised.

The remainder of this paper will first describe the relevant parts of the C semantics in section 2. In section 3, we explain why what might initially appear to be non-deterministic is in fact deterministic, and outline the overall proof strategy. The proof is explained in more detail in sections 4 and 5, and section 6 concludes.

2 The semantics of C expressions

Cholera models the semantics of C's expressions with a reduction style operational semantics using a relation $\rightarrow_e$ such that $\langle e_0, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle$ holds when an expression e_0 in state σ_0 can take a step, becoming a new expression e, and with the state changing to state σ. States σ, σ_0, σ' etc. embody not just the usual mapping from variables to values, but also information about the program environment, and pending side effects. We use $\rightarrow_e^*$ to denote the reflexive and transitive closure of the single step relation.

There are two principal sources of non-determinism in the semantics. These are the rules for the evaluation of binary expressions and the way in which side effects are applied. The following two rules illustrate the first:

$$\frac{\langle e_1, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle}{\langle e_1 \odot e_2, \sigma_0\rangle \rightarrow_e \langle e \odot e_2, \sigma\rangle} \qquad \frac{\langle e_2, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle}{\langle e_1 \odot e_2, \sigma_0\rangle \rightarrow_e \langle e_1 \odot e, \sigma\rangle}$$

Here $\odot$ stands for all of C's arithmetic operators, but not for the logical operators `&&` and `||`, nor the comma operator, nor the assignment operators. The first rule says that if the first argument to a binary operator can take a step in the semantics, then so too can the containing expression. The non-determinism enters because both rules apply at all times, meaning that an expression is *not* constrained to evaluate its operands in any particular order, and may even interleave the evaluation of its operands. In the presence of side effects and changes to the state, this is potentially a significant source of non-determinism.

The second source of non-determinism in the semantics is its handling of side effects. Side effects are generated by the evaluation of assignment expressions and the various increment and decrement operators (`++` and `--`). These operators have as their side effects the writing of values into memory, but this does not necessarily happen immediately. Instead, side effects are applied at arbitrary times and in any order, subject only to the constraint that all pending side effects be applied before the next *sequence point*. Sequence points occur at certain well-marked stages in expression evaluation, such as after the complete evaluation of the first argument to the logical operators `&&` and `||`. The rule for side effect application is:

$$\frac{\eta \text{ is pending in } \sigma}{\langle e, \sigma\rangle \rightarrow_e \langle e, \texttt{apply\_se}(\sigma, \eta)\rangle}$$

where $\texttt{apply\_se}(\sigma, \eta)$ denotes the state resulting from the application of side effect η to state σ, with the appropriate changes made (memory updated, and η removed from the pending side effects).

2.1 Constraints on expression evaluation

The above description of expression evaluation suggests a chaotic picture. A naive interpretation would suggest that the evaluation of

```
v + v++ + v + v++
```

with `v` initally 3, could yield any value in the range 12–17. However, the language definition imposes severe constraints on the way in which expressions can evaluate, and in fact, this expression is undefined.

The constraint is that "between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored." [3, §6.3] (For our purposes, an object is best understood as simply a part of memory.) Violation of this constraint results in undefinedness.

It is worth noting that this is a constraint on the dynamic behaviour of the program. Though the expression given above involving `v` will necessarily be undefined because it both refers to and updates the object denoted by `v`, it is not clear whether or not this is true of `*p + (i = 1)`, say, as it is impossible in general to determine whether or not `*p` (a dereferencing of pointer variable `p`) will refer to `i`. Finally, note that the second sentence quoted above allows references to take place if they occur on the right-hand of an assignment expression and the references are to

the object being modified by the assignment. For example, this clause allows `i = i + 1`.

A formal semantics of C must model this constraint as well as the more obvious rules given earlier. To do this, Cholera keeps track of three state components:

- the pending side effects
- those parts of memory which have been updated (the "`update_map`")
- those parts of memory which have been referred to (the "`ref_map`")

The pending side effects component is a multi-set, or bag, as the same side effect might occur twice in a given evaluation. The `update_map` is a set of addresses, as no evaluation will be allowed to update the same location twice. The `ref_map` is another bag, as multiple references to the same location can legitimately occur. As we shall see, we need to know how many references were made to a particular location, not just whether or not something has been referred to.

There are four different ways in which these components can change in the evaluation of an expression.

- When a non-array lvalue becomes a value, the `ref_map` is increased to reflect the reference of the object denoted by the lvalue.[2] If the part of memory referred to is in the `update_map`, this causes undefinedness.
- When a side effect is applied, it is removed from the pending side effects bag, and the `update_map` is increased, recording the fact that part of memory has just been changed. If that part of memory has already been updated or referred to, this causes undefinedness.
- When an assignment completes its evaluation, a side effect to update the appropriate part of memory with a new value is added to the pending side effects bag. Assignment expressions keep track of references made on their right hand sides, and those that were to the object to be updated are removed from the `ref_map`. Failure to do this would cause the side effect created as a result of evaluating `i = i + 1` to clash with the reference to `i` on the expression's RHS.
  Because the `ref_map` records a count of the number of times a piece of memory has been referred to, this deletion of references may still leave references recorded. Using only a set for `ref_map` would allow

```
i + (i = i + 1)
```

[2] Array lvalues become pointers to their first element; this transformation does not require a reference to memory.

to avoid revealing its undefined nature. A possible evaluation would have the `i` on the assignment's RHS remove the record of a previous reference to `i` on the LHS of the addition.
– When the pending side effects bag is empty, and a sequence point is reached in an expression's syntax, the `ref_map` and `update_map` are "zero-ed", thereby allowing a new sequence of reference and updates in the next phase of execution. If a sequence point is reached, and the bag of pending side effects is not empty, it will need to be emptied before the next stage of the expression can be evaluated.

3 Intuition and proof outline

This may have already suggested that C's expression semantics, though superficially full of non-determinism, is actually so seriously constrained that expressions can only evaluate in one way, whether this be to one valid result, or to undefinedness. Here we suggest why this is the case, and sketch the form of the proof that is to come.

Ignoring for the moment the fact that side effects are not necessarily applied immediately nor in order, one can think of the various sub-expressions of a greater expression as parallel processes running simultaneously and sharing memory. Clearly, the behaviour of these processes is solely dependent on the parts of memory that they reference. But this implies that the processes can't affect each other: a change to a piece of memory by one process that another references is forbidden by the constraints spelled out in the previous section.

If a sub-expression can't affect the parts of memory that another depends on, and *vice versa*, then their evaluation must proceed entirely deterministically. Conversely, if shared memory *is* updated illegally, then undefinedness must result.

The fact that side effects are not applied immediately is also seen to be irrelevant. All side effects will come to be applied eventually, as reaching a sequence point requires this, and at the minimum, there is a sequence point at the end of the evaluation of all expressions that appear within statements. Though an update may come quite late, the constraints forbid the updating of memory that has been referred to as much as they forbid reference of updated memory.

Nonetheless, there is still a problem with the above intuition: it ignores the effect of sequence points that appear within an expression. Consider the following expression:

```
x + ((x = 3), 4)
```

Given what we have seen so far of the semantics, it would appear that this expression should be genuinely non-deterministic. The comma operator on the right is a sequence point, so if an evaluation were to proceed by first evaluating `x = 3`, reaching the sequence point, clearing the `update_map`, and then proceeding with the rest of the expression, it should go on to give a result of 7.

On the other hand, if the lone `x` on the left were to be evaluated first, then the subsequent assignment expression (necessarily the next thing to be evaluated) would cause undefinedness, because it would update an object which had already been referenced.

In fact, a subtle argument about this case forces the conclusion that the expression is necessarily undefined.[3] An official response by the Standards committee to a public query (a "Defect report") [4, #117] makes it clear that if it is possible for an expression to exhibit undefined behaviour (there might be an order of evaluation that does this, for example), then the whole expression *is* undefined.

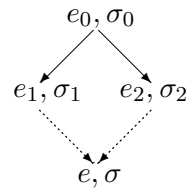
Cholera does model this requirement, but is forced to do so at the level above the definition of $\rightarrow_e$. We add a rule to the effect that a given, defined, reduction sequence is only part of the semantics if there doesn't exist any other sequence which makes the behaviour undefined. However, this additional detail in the semantics is difficult to reason about, so we choose to examine those expressions which are free of internal sequence points. Unless otherwise stated, all results stated here will be for expressions that are free of internal sequence points.

Our eventual determinism result for sequence point free expressions then naturally holds of expressions where the only sequence points are present at the top level (such as in `x || (y && z)`). This is because such an expression has deterministic sub-expressions, and these must be evaluated in the order dictated by the presence of the sequence points, giving an overall behaviour which must also be deterministic.

3.1 Proof outline

We should like to demonstrate determinism by showing a *diamond property* for all of the possible reductions that an expression might undergo. Graphically, this amounts to showing that in all situations we can find reductions to fill in the dashed lines below:

[3] My thanks to Mark Brader for explaining this to me.



It follows that if this can be shown for single steps of a reduction relation, then that reduction system must be confluent. Unfortunately, this property does *not* hold in general for C. In particular, reductions that involve undefined behaviour tend to invalidate further reductions, and if the reduction to $\langle e_1, \sigma_1 \rangle$, say, caused undefined behaviour then there is no guarantee that a reduction analogous to the one taken to get to $\langle e_2, \sigma_2 \rangle$ should be possible.

Therefore, the first step in attacking the proof is to divide it into two parts. First we demonstrate confluence for evaluations which terminate normally, i.e., those which yield a value and which apply all of the side effects generated in the course of the expression evaluation. Then we show that if an evaluation sequence exists which leads to undefined behaviour, this undefinedness can not be escaped, and that all states reachable from the initial one must necessarily either be undefined themselves, or still admit the possibility of becoming undefined in one or more steps.

This second result makes it clear that a normal terminating evaluation and an undefined one can not both begin from the same initial state. We reason as follows: assume that such a situation exists. Then our second result states that it is possible to reach undefinedness from the final state of the normal evaluation. But if it is a final state, then it can not take any more steps, and it is not in an undefined state itself because it has yielded a proper value. Thus we have a contradiction and an assurance to the effect that all evaluations are in fact deterministic.

4 Successful evaluations

Even with the above assumption that our reduction sequences do not become undefined, the task of proving determinism for expression evaluation is quite complicated. In particular, the system as described is made difficult to reason about by the fact that side effect applications and other forms of reduction can intermingle. The first stage of our proof is

to demonstrate that side effect applications can all be postponed to the end of an evaluation sequence without affecting the result.

This should be clear from the constraints described earlier: if a side effect application were to make a difference, a subsequent reference to memory would need to look at some part of memory that the side effect had changed; but this is precisely one of those situations forbidden (a reference to updated memory) and would lead to undefinedness, contradicting our earlier assumption.

The proof proceeds by first showing that side effect applications and other reductions can commute.

Lemma 1. *For all expressions e_0, e_1, for all states σ, σ_0, σ_1, and for all side effects η, if η is pending in σ, with $\sigma_0 = \texttt{apply\_se}(\sigma, \eta)$ (i.e., σ_0 is the state that results from applying η to σ), and $\langle e_0, \sigma_0\rangle \rightarrow_e \langle e, \sigma_1\rangle$ then there exists a state σ' such that $\langle e_0, \sigma\rangle \rightarrow_e \langle e, \sigma'\rangle$, η is pending in σ' and $\sigma_1 = \texttt{apply\_se}(\sigma', \eta)$.*

This is a straightforward rule induction on the inductive definition of $\rightarrow_e$. Another induction readily extends this to allow side effect applications to be pushed past any number of other expression reduction steps. Using this, we then induct on the number of reductions to prove our "separation theorem":

Theorem 1 (Separation). *For all expressions e_0, e, and for all states σ_0, σ, if $\langle e_0, \sigma_0\rangle \rightarrow_e^* \langle e, \sigma\rangle$, then there exists a state σ' and a sequence of side effects $\eta_1 \ldots \eta_n$ where both the* update_map*s and memory contents of σ_0 and σ' are the same, and $\langle e_0, \sigma_0\rangle \rightarrow_e^* \langle e, \sigma'\rangle$ and σ is the result of applying the side effects $\eta_1 \ldots \eta_n$ to σ'.*

(Note that the final value e is present after the expression reduction steps, and before the side effect applications begin. This is because these later applications can not change the value that an expression yields.)

We now consider the $\rightarrow_e$ relation as the union of two components: reductions where no side effect applications occur, and reductions that are exclusively side effect applications. Let us use $\rightarrow_E$ for the former and $\rightarrow_A$ for the latter so that $\rightarrow_e = \rightarrow_E \cup \rightarrow_A$. Confluence for both $\rightarrow_E$ and $\rightarrow_A$, together with the separation theorem imply confluence for $\rightarrow_e$ as follows:

1. Consider two reduction sequences starting at $\langle e_0, \sigma_0\rangle$ that both complete normally. One is to $\langle e_1, \sigma_1\rangle$ and the other is to $\langle e_2, \sigma_2\rangle$.

2. By the separation theorem, both reduction sequences can be separated into two phases, with intermediate points $\langle e_1, \sigma_1' \rangle$ and $\langle e_2, \sigma_2' \rangle$, such that $\langle e_0, \sigma_0 \rangle \rightarrow_E^* \langle e_1, \sigma_1' \rangle$ and $\langle e_1, \sigma_1' \rangle \rightarrow_A^* \langle e_1, \sigma_1 \rangle$ (similarly for e_2 etc.)
3. Because e_1 and e_2 represent completed evaluations, they must be values. As $\rightarrow_A$ only applies side effects, it doesn't change expressions. Thus the states reached by $\rightarrow_E^*$ must be terminal with respect to it. Then if $\rightarrow_E$ is confluent, these intermediate states are actually the same.
4. Now we have two reduction sequences involving $\rightarrow_A^*$ from the same starting point. As $\rightarrow_A$ is also confluent, the final states are necessarily identical.

Given this result, we need only prove that $\rightarrow_E$ and $\rightarrow_A$ are confluent.

4.1 Confluence for $\rightarrow_E$

We establish confluence for $\rightarrow_E$ by demonstrating a diamond property for single steps of the relation.

Before beginning a proof such as this, it is instructive to consider parallels with the similar task that one faces in attempting to prove confluence for the λ-calculus. There, things are somewhat complicated by the fact that a reduction in the RHS of a β-redex may have to be matched by many repetitions of essentially the same reduction in an alternative branch where the RHS has been substituted into the body of the LHS. This doesn't happen in the Cholera semantics, where substitution doesn't arise.

However, the λ-calculus is at least entirely syntax-directed; if a redex is present, then the reduction can always take place, and its result will always be the same. Reductions in the λ-calculus can be said to ignore their context. This is not the case in Cholera where the accompanying state, an ever-present and varying context, can affect reductions. This is not just a matter of different values for variables affecting the value of an expression, but more significant: a state with a large `update_map` may make a reduction that would otherwise turn a variable into a value instead produce undefinedness.

With this motivation behind us, the first stage in our proof will be to characterise the degree to which states can vary and yet still produce the same reduction for a given piece of syntax. Furthermore, because

expression reductions affect the state[4], we want to characterise the way in which this happens, so that, ultimately, we will be able to state that reduction x can reduce in the same way both before and after reduction y.

Theorem 2 (Reduction characterisation). *If* $\langle e_0, \sigma_0 \rangle \rightarrow_E \langle e, \sigma \rangle$ *then there exists a function* f *characterising the reduction, such that* $f(\sigma_0) = \sigma$*, and for all* σ_0' *which are "no more restrictive" than* σ_0*, then* $\langle e_0, \sigma_0' \rangle \rightarrow_E \langle e, f(\sigma_0') \rangle$.

The meaning of "no more restrictive" above turns out to be rather detailed in its expression, really suitable only for the consumption of a theorem prover. In essence it requires that the `update_map` be no bigger in σ_0' than it is in σ_0, but there are also are a number of conditions required of both the initial states and the expressions involved. One of these is that e_0 be well-typed. Another is that e not be undefined; computations that do allow e to become undefined are discussed in section 5.

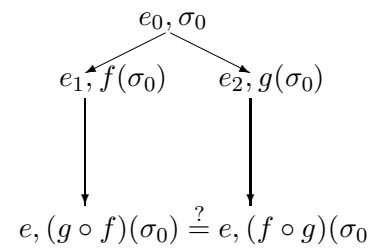
We also have the following important lemma, which like the previous is established by induction over the reduction relation.

Lemma 2 (Reduction preconditions preserved). *If* $\langle e_0, \sigma_0 \rangle \rightarrow_E \langle e, \sigma \rangle$*, then* σ *is no more restrictive than* σ_0 *in the sense of theorem 2.*

Now we can prove the diamond property for $\rightarrow_E$ relatively straightforwardly. Again an induction is required over the reduction relation. The inductive or "sub-expression" cases, where we have two reductions within the same sub-expression, are handled by the inductive hypotheses, so it is just the cases where an expression form admits two reductions in different sub-expressions which prove difficult. This includes both the normal binary operators, and also assignment, which needs to be treated separately because unlike the other operators, it adds a side effect to those pending.

In such a situation, our reduction characterisation and reduction preconditions results tell us immediately that a "diamond" of four sides can be constructed. If the functions required to exist by the first result are f and g, then the diagram looks like:

[4] Though $\rightarrow_E$ holds `update_map`s and thus memory constant, we will still get new side effects being added to the queue of those pending, and as objects are referred to, `ref_map`s also increase.



The question then remains as to whether or not f and g will commute. They do in fact, as each does little more than specify the additions to the starting state's `ref_map` and pending side effects. Addition on bags being commutative, the result follows.

4.2 Confluence for $\rightarrow_A$

The second requirement of the proof of is to show that the $\rightarrow_A$ relation is confluent. We show this by demonstrating a diamond property. This is a considerably simpler task than for $\rightarrow_E$.

Recall that we are performing reductions in a context where all of the side effects can be applied successfully, resulting in normal termination with a value. This implies that no pair of pending side effects affect overlapping parts of memory. We show this by contradiction. One of the side effects must have been applied first. Subsequent to this application, the other side effect can not have been applied because this would result in undefined behaviour (two updates of the same part of memory). But if the second side effect is not applied, then the final state must still have side effects pending, which also contradicts our assumption, because a normal termination is a sequence point, by which state all side effects must have been applied.

So, all of the side effects affect different parts of memory, and can therefore be applied independently of one another. The required diamond property is an immediate consequence of this.

5 Undefined evaluations

We begin by defining *state safety*. A state is *safe* if none of its pending side effects conflict neither with each other (i.e., do not affect overlapping parts of memory), nor with the state's `ref_map` and `update_map`. It should be clear that a state which is safe can apply all of its side effects without becoming undefined. The converse is the basis of our first lemma in this section.

Lemma 3 (Finite and unsafe states can become undefined). *If a state σ_0 is both unsafe and has a finite bag of pending side effects, then for all e_0 there exists a reduction sequence such that $\langle e_0, \sigma_0\rangle \rightarrow_e^* \langle \mathcal{U}\rangle$, where $\mathcal{U}$ represents undefinedness.*

Also, for all e_0, e, σ_0 and σ, if σ_0 is unsafe, and $\langle e_0, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle$, then σ is also unsafe.

Cholera represents undefinedness arising as a result of expression evaluation (e.g., division by zero, or a reference to a variable already updated) by replacing the offending expression with $\mathcal{U}$ in the syntax tree and then letting this "bubble" its way to the top of the tree. This can not be prevented.

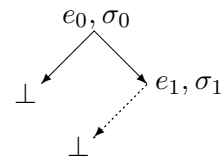
Lemma 4 (Undefined sub-expressions can always ascend). *If an expression e_0 contains $\mathcal{U}$ as a sub-expression, then for all σ_0 there exists a reduction sequence such that $\langle e_0, \sigma_0\rangle \rightarrow_e^* \langle \mathcal{U}\rangle$.*

Also, for all e_0, e, σ_0 and σ, if e_0 has an undefined sub-expression, and $\langle e_0, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle$, then e must also have an undefined sub-expression (where e itself may be that undefined sub-expression).

These two results (neither of which is particularly surprising) make it clear that a large class of expression-state pairs, those which are unsafe or which have undefined subexpressions, though not necessarily "fully undefined", might as well be. We shall refer to such states as *effectively undefined*. Though a state's being effectively undefined may not seem such a strong claim initially, the condition preservation clauses of the lemmas above should make it clear that an effectively undefined state is one which can never yield a value. In conjunction with the fact that all sequence point free expressions must terminate[5], we can see that effectively undefined means "will necessarily become undefined".

Our next theorem is more significant. We wish to show that if a reduction occurs which makes something effectively undefined, when it was not effectively undefined before, then if one takes a different step from the same initial state, the result will either be effectively undefined, or it will retain the ability to make a reduction to an effectively undefined state. This can be represented as a "broken" diamond:

[5] Sequence point free expressions do not include function applications.



Another analogy is that of the cliff-edge. Over the edge lies effective undefinedness. Once one reaches the edge, one can walk along it, but while it may be possible to avoid falling over the edge for some indeterminate length of time, it is not possible to move away. The proof proceeds in a similar way to that of the proof of the confluence of $\rightarrow_E$.

While the inductive cases are straightforward, we need to cope with the fact that the reduction from $\langle e_0, \sigma_0 \rangle$ to $\langle e_1, \sigma_1 \rangle$ might involve a reduction in a sub-expression unrelated to that which produced the undefinedness. Inside $\langle e_1, \sigma_1 \rangle$ we want to have a reduction occur that is analogous to the one that produced undefinedness from $\langle e_0, \sigma_0 \rangle$. We do this by again establishing a reduction characterisation result, and by demonstrating that reductions preserve this.

In this case, the characterisation is essentially that an analogous reduction to undefinedness can occur in any state that is at least as restrictive as the original. This condition is preserved both by $\rightarrow_E$ and $\rightarrow_A$.

We then do an induction of the number of steps along the cliff's edge to produce:

Theorem 3 (The cliff's edge). *For all e_0, σ_0, if $\langle e_0, \sigma_0 \rangle \rightarrow_e \langle e_1, \sigma_1 \rangle$ and $\langle e_1, \sigma_1 \rangle$ is effectively undefined, then for all e_2 and σ_2 such that $\langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle e_2, \sigma_2 \rangle$, there exists e' and σ' such that $\langle e_2, \sigma_2 \rangle \rightarrow_e \langle e', \sigma' \rangle$, and $\langle e', \sigma' \rangle$ is effectively undefined.*

We still need to add one more diamond property. This is a surprisingly easy proof as it does not require an induction over the meaning relation. Instead the characterisation functions and our lemma (1) that $\rightarrow_E$ and $\rightarrow_A$ commute combine to give:

Theorem 4 (A diamond property for $\rightarrow_E$ and $\rightarrow_A$). *For all e_0, e_1, e_2 σ_0, σ_1, σ_2: if $\langle e_0, \sigma_0 \rangle \rightarrow_E \langle e_1, \sigma_1 \rangle$ and $\langle e_0, \sigma_0 \rangle \rightarrow_A \langle e_2, \sigma_2 \rangle$ and both $\langle e_1, \sigma_1 \rangle$ and $\langle e_2, \sigma_2 \rangle$ are not effectively undefined, then there exist e and σ (possibly effectively undefined), such that $\langle e_1, \sigma_1 \rangle \rightarrow_A \langle e, \sigma \rangle$ and $\langle e_2, \sigma_2 \rangle \rightarrow_E \langle e, \sigma \rangle$.*

The final proof is now possible. We wish to show that if a reduction sequence takes an initial state ($\langle e_0, \sigma_0 \rangle$) to undefinedness, then all other

possible destinations from the same starting point retain this possibility. In essence, we exploit the possibility of completing a confluent diamond on the cliff-tops.

1. We have a reduction sequence from $\langle e_0, \sigma_0 \rangle$ to undefinedness. Let $\langle e_1, \sigma_1 \rangle$ be the last state in this sequence not effectively undefined.
2. We have another reduction sequence to $\langle e_2, \sigma_2 \rangle$, and by assumption this is not effectively undefined.
3. Therefore, using all three of our diamond properties for $\rightarrow_E$ and $\rightarrow_A$, we have a common possible destination for both $\langle e_1, \sigma_1 \rangle$ and $\langle e_2, \sigma_2 \rangle$. Call this $\langle e, \sigma \rangle$.
4. Having come along the cliff's edge from $\langle e_1, \sigma_1 \rangle$, $\langle e, \sigma \rangle$ must still be on the edge, thereby retaining the possibility of a reduction to an effectively undefined state, if it is not an effectively undefined state already.
5. Effectively undefined states all allow for a reduction sequence to "full" undefined-ness, so $\langle e_2, \sigma_2 \rangle$ must do so as well by virtue of being able to reduce to $\langle e, \sigma \rangle$.

6 Conclusion

The fact that we have ended with proofs of diamond properties for $\rightarrow_E$, $\rightarrow_A$ and $\rightarrow_E$ *vs.* $\rightarrow_A$ may suggest that the rather specialised proof strategy used in section 4.1 might as well have been subsumed into an all-encompassing proof of confluence for the whole meaning relation. In particular, it is easy to see with hindsight that demonstrating a diamond property, where neither reduction is to an effectively undefined destination, would have been reasonably straightforward. Nonetheless, the only theorem that becomes redundant in this alternative proof is the separation result (theorem 1). All the other results given are necessary parts of either proof.

It is extremely important that this work was built on the support provided by mechanical theorem proving (HOL, in this case). It would have been unimaginable without that support. The proof script for proving this result is almost 6000 lines of SML code (excluding comments). This work is thus a demonstration of both the importance and utility of mechanised theorem-proving. The mechanisation of the semantics ensures that one can be sure of one's results, and that no details have been overlooked. In this work, the diamond proofs in question involved analysis of many (approximately 200) different cases corresponding to a pair-wise examination of all the possible ways in which all possible expressions might

evolve. Such a proof done by hand would be inevitably subject to question because of the high possibility of error. With HOL's help, the possibility of error has been eliminated.

This work is also valuable because it demonstrates an interesting result about the programming language C. This in turn is a demonstration that the practical formalisation of programming language semantics is not an impossible dream. In [8], Ritchie says "the C standard did not attempt to specify formally the language semantics, and so there can be dispute over fine points". In the formal setting provided by Cholera, fine points are no longer the subject of dispute: not only does the language gain an unambiguous specification, it is also possible to state the definition's consequences with certainty.

References

1. M. J. C. Gordon and T. Melham. *Introduction to HOL: a theorem proving environment.* Cambridge University Press, 1993.
2. Matthew Hennessy. *The semantics of programming languages.* John Wiley and Sons, 1990.
3. *Programming languages – C*, 1990. ISO/IEC 9899:1990.
4. ISO committee JTC1/SC22/WG14. Record of responses. Available from `ftp://ftp.dmk.com/DMK/sc22wg14/rr/`.
5. Michael Norrish. Derivation of verificiation rules for C from operational definitions. In J. von Wright, J. Grundy, and J. Harrison, editors, *Supplementary proceedings of TPHOLs '96*, number 1 in TUCS General Publications, pages 69–75. Turku Centre for Computer Science, August 1996.
6. Michael Norrish. An abstract dynamic semantics for C. Technical Report 421, Computer Laboratory, University of Cambridge, May 1997.
7. Michael Norrish. *C formalised in HOL.* PhD thesis, Computer Laboratory, University of Cambridge, 1998. Submitted August, 1998.
8. D. M. Ritchie. The development of the C language. *ACM SIGPLAN Notices*, 28(3):201–208, March 1993.