Computation Systems with Restartable Computations

Carolyn Talcott
Stanford University
clt@sail.stanford.edu
Richard W. Weyhrauch
Ibuki Inc.
rww@ibuki.com

1. Introduction

In this note we introduce a notion of computation system that provides a *restartable stepper* semantics for programs. Our notion of computation system is motivated by two goals. The first is to give an axiomatic characterization of restartable computation. The second is to provide a basis for giving a finitistic semantics to first order logic in which classical reasoning is preserved.

Stepper semantics is based on the notion of resource limited computation steps. A step gives the next stage of a computation. If the computation is complete, then a value can be extracted. Otherwise it provides information on how to continue the computation. Using the stepper semantics, the attempt to compute a value can simply be abandoned, or it can be suspended with the possibility of restarting later. The semantics cannot distinguish between a program that will never be able to compute a value (diverging or stuck) and a computation that was abandoned simply because the stepper used up its currently available resources. The use of the stepper semantics as the basis of a computation system gives us the tools for developing a semantics for first order logic that allows reasoning to be interrupted without breaking or exiting the system.

We present the notion of computation system informally as partial structures of a particular similarity type that satisfy certain axioms (see [2] for for a definition of similarity types and partial structures). We give as an example a finite partial structure which is the worlds smallest computation system.

2. Computation systems as partial structures

A *computation system* is a partial structure, $Csys$, that has as its universal sort a set, $CsysU$, of computational entities (the computational universe). The basic sorts of computational entities of $Csys$ are:

Pgm, a set of programs;

$Data$, a set of data to be manipulated by programs;

Cmp, a set of computations;

$Done$, the subset of Cmp consisting of the completed computations;

$NcCmp$, the subset of Cmp consisting of computations that never complete;

$Labable$, the computational entities that may be labelled (placed in an environemt for access by programs);

$Resource$, a set of computing resources;

Lab, a set of labels;

$ReLab$, relabelling maps—finite bijections on labels; and

Env, a set of computation environments—finite maps from labels to entities that can be labelled.

We let P range over Pgm, d range over $Data$, $resource$ range over $Resource$, lab range over Lab, and env range over Env.

In addition to these basic computational entities, we need to form finite lists. We let $ListOf[Data]$ be the sort of lists of data elements used as arguments, let $ListOf[Cmp]$ be the sort of lists of computations, and let $ListOf[Lab]$ be the sort of lists of labels. We let $args$ range over $ListOf[Data]$, $cmps$ range over $ListOf[Cmp]$, and $labs$ range over $ListOf[Lab]$.

The relations of $Csys$ are:

$\bowtie$, a binary relation expressing a notion of compatability for computations; and

run, a binary relation giving the result of a computation if it completes.

The functions of $Csys$ are:

$const$, a unary function mapping data to (completed) computations (the constant computations).

$result$, a unary function mapping completed computations to the resulting value;

apply, a function mapping a program, an environment, and a list of computations to a computation;

call, a function mapping a program, an environment, and an argument list to a computation;

rcompose, a binary function that composes resources, allowing finite sequences of resources to be represented by a single resource;

labsOf, a unary function mapping programs and computations to finite sets of labels;

relabApp, a binary function that applies relabelling maps to programs, and computations;

step, a binary function that takes a resource and a computation and returns a computation.

The environment argument to apply and call include information such as function declarations, global constants, etc. needed to interpret the program. The resource argument of step bounds the amount of work that the stepper can do.

The individuals of *Csys* are:

yes and *no*, are distinct distinguished elements of *Data* used as booleans (binary answers to questions);

nc-cmp is a never completing computation, and *nc-pgm*, a never completing program;

mt-rsc is an empty resource, it allows no work to be done; and

mt-env is an empty environment, one with no information.

We also have the emptylist, *MtList*, the empty set *MtSet*, and the empty map, *MtMap*. And the basic operations on lists, sets, and maps.

The similarity type, τ_{Csys}, of a computation system partial structure is implicit in the above description. We require that the following properties hold for a partial structure, *Csys*, of similarity type τ_{Csys} to be a computation system.

C1. *const* produces completed computations on arguments.

$$Done(const(d)) \quad \text{and} \quad result(const(d)) = d$$

C2. Stepping does not change completed computations.

$$Done(cmp) \supset step(resource, cmp) = cmp$$

C3. Stepping with the empty resource is a no-op and stepping distributes over with resource composition.

$$step(mt\text{-}rsc, cmp) = cmp$$

$$step(rcompose(resource_0, resource_1), cmp) =$$
$$step(resource_1 step(resource_0, cmp))$$

C4. *NcCmp* is the set of *never completing* computations, and *nc-cmp* is never completing.

$$NcCmp(cmp) \equiv (\forall resource)\neg(Done(step(resource, cmp)))$$

$$NcCmp(nc\text{-}cmp)$$

C5. Computations of *nc-pgm* are never completing.

$$NcCmp(apply(nc\text{-}pgm, env, cmps))$$

C6. Computations are coherent in the sense that stepping a computation using different resources gives compatable results – if both are terminated then they yield the same result.

$$\begin{aligned}&(\forall\ resource_0, resource_1)(\\ &\quad Done(step(resource_0, cmp)) \wedge\\ &\quad Done(step(resource_1, cmp)) \supset\\ &\qquad result(step(resource_0, cmp))\\ &\qquad\quad =\\ &\qquad result(step(resource_1, cmp)))\end{aligned}$$

C7. The defining axiom for compatability relation on computations, $cmp_0 \bowtie cmp_1$, generalizes axiom C6.

$$\begin{aligned}cmp_0 \bowtie cmp_1 \equiv (\forall\ &resource_0, resource_1)(\\ &Done(step(resource_0, cmp_0)) \wedge\\ &Done(step(resource_1, cmp_1)) \supset\\ &\quad result(step(resource_0, cmp_0)) =\\ &\quad result(step(resource_1, cmp_1)))\end{aligned}$$

In the presence of C7, axiom C6 can be expressed as self-compatability. An additional simple consequence of C6 and C7 is that stepping preserves compatability. A special case of this is that the result of stepping a computation is compatable with the computation. These consequences are made precise in the following lemma.

Lemma (compat):

(0) $cmp \bowtie cmp$

(1) $cmp_0 \bowtie cmp_1 \supset step(rsc_0, cmp_0) \bowtie step(rsc_1, cmp_1)$

(2) $cmp \bowtie step(rscs, cmp)$

An interesting subclass of never completing computations are those that are stepper fixed points. These are computations, cmp such that

$$\neg(Done(cmp)) \wedge \\ (\forall resource \in Resource)(\forall env \in Env) \\ (step(resource, cmp) = cmp)$$

C8. The function $call$ is defined by:

$$call(P, env, args) = apply(P, env, cmps)$$

where $cmps$ is the unique list of computations satisfying

$listLen(cmps) = listLen(args)$, and

$listElt(cmps, i) = const(listElt(args, i))$, for $1 \leq i \leq listLen(args)$.

C9. The function run is defined by:

$$run(cmp, u) \equiv \\ (\exists resource)(Done(step(resource, cmp)) \wedge \\ result(step(resource, cmp)) = u)$$

Lemma (unicity): The run relation has the unicity property, namely there is at most one u such that $run(cmp, u)$.

E0. Labelables include programs, arguments, computations ???

E1. Environments are finite maps from labels to labelables:

$$Env = Map[Lab, Labable]$$

We extend relabelling maps to environments by

$$relabApp(d, relab) = d$$

$$relabApp(relab, env) = \lambda l \in relab(\mathrm{Dom}(()env)).relabApp(relab, env(relab^{-1}))$$

Labelling For $x \in Pgm \cup Cmp$

$$labsOf(relabApp(relab, x)) = \{mapApp(relab, lab) \mid labinlabsOf(x)\}$$

$$relabApp(relab_1 \, cmps relab_0, x) = relabApp(relab_1, relabApp(relab_0, x))$$

$$mapDom(relab) \cap labsOf(x) = MtSet \supset relabApp(relab, x) = x$$

$$step(resource, relabApp(relab, cmp))$$

$$= relabApp(step(resource, cmp))$$

$$Done(cmp) \equiv Done(relabApp(relab, cmp))$$

$$Done(cmp) \supset result(relabApp(relab, cmp)) = result(cmp)$$

$$apply(relabApp(relab, P), relabApp(relab, env), relabApp(relab, cmps))$$

$$= relabApp(relab, apply(P, env, cmps)$$

Where relabelling is extended to lists via mapping along the list.

Lemma (relab):

$$cmp \bowtie relabApp(relab, cmp)$$

$$cmp_0 \bowtie cmp_1$$

$$\equiv relabApp(relab, cmp_0) \bowtie relabApp(relab, cmp_1)$$

3. The Smallest Computation System

As a first example we define the *smallest* computation system $Csys_{\mathrm{small}}$. This is the partial structure given by:

$$CsysU = Cmp = \{\mathtt{no}, ?, \mathtt{yes}\}$$

$$Data = Done = \{\mathtt{no}, \mathtt{yes}\}$$

$$Pgm = Resource = Env = \{?\}$$

$$yes = \mathtt{yes}$$

$$no = \mathtt{no}$$

$$nc\text{-}pgm = nc\text{-}cmp = mt\text{-}env = mt\text{-}rsc = ?$$

const and *result* are identity functions,

apply has range $\{?\}$,

$rcompose(?, ?) = ?$, and

$step(?, ?, d) = d$ for any computation d.

Note that this partial structure is both total and finite. Otherwise, this is a pretty uninteresting computation structure, but its shows that such partial structures exists, and it is adequate for building a minimal simulation structure (as described in [3]).

4. References

[1] R. W. Weyhrauch and C. L. Talcott. FOL home page, 1995. URL = http://www-formal.stanford.edu/FOL/home.html.

[2] R. W. Weyhrauch and C. L. Talcott. Set-theoretic FOL systems, 1995. URL = http://www-formal.stanford.edu/FOL/home.html.

[3] R. W. Weyhrauch and C. L. Talcott. Fol contexts – the data structures, 1997. URL = http://www-formal.stanford.edu/FOL/home.html.