Technical Report http://hdl.handle.net/2142/10790, January 2009

Matching Logic — Extended Report

Grigore Roşu
University of Illinois at Urbana-Champaign
grosu@illinois.edu

Wolfram Schulte
Microsoft Research Redmond
schulte@microsoft.com

Abstract

Hoare logics rely on the fact that logic formulae can encode, or specify*, program states, including environments, stacks, heaps, path conditions, data constraints, and so on. Such formula encodings tend to lose the structure of the original program state and thus to be complex in practice, making it difficult to relate formal systems and program correctness proofs to the original programming language and program, respectively. Worse, since programs often manipulate mathematical objects such as lists, trees, graphs, etc., one needs to also encode, as logical formulae, the process of identifying these objects in the encoded program state.*

This paper proposes matching logic*, an alternative to Hoare logics in which the state structure plays a crucial role. Program states are represented as algebraic datatypes called* (concrete) configurations*, and program state specifications are represented as configuration terms with variables and constraints on them, called* (configuration) patterns*. A pattern* specifies *those configurations that* match *it. Patterns can* bind *variables to their scope, allowing both for* pattern abstraction *and for expressing* loop invariants*.*

Matching logic is tightly connected to rewriting logic semantics (RLS): matching logic formal systems can systematically be obtained from executable RLS of languages. This relationship allows to prove soundness *of matching logic formal systems w.r.t. complementary, testable semantics. All notions are exemplified using* KernelC*, a fragment of C with dynamic memory allocation/deallocation.*

1. Introduction

Program reasoning approaches are conventionally based on an axiomatic semantics of the programming language as a formal system deriving (partial or total) correctness triples $\{\varphi\}\,\texttt{PGM}\,\{\varphi'\}$, also called Hoare triples, where the precondition φ and the postcondition φ' are formulae in some logic of choice. The underlying intuition is that logical formulae can symbolically encode program states, so $\{\varphi\}\,\texttt{PGM}\,\{\varphi'\}$ relates program states before the execution of PGM to corresponding states resulting after the execution of PGM.

The logic originally used by Hoare [7] for formulae was first-order logic (FOL). Driven by practical needs, researchers have extended FOL with various domains and decision procedures, as well as with means to recursively define predicates. However, as rightfully noted by Reynolds and O'Hearn among many others [15, 8], in spite of three decades of study, FOL-based approaches still suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size. For example, even proving that a small C program reverses a list (each list node contains a value and a pointer to the next node) is a highly non-trivial task for FOL-based approaches [15].

The last decade has seen an increasing interest in extensions of FOL more suitable to reason about shared data, such as TVLA [10] or separation logics [15, 8]. For the latter, e.g., new logical connectives are added to FOL to specify data separation, e.g., the separating conjunction: $\varphi_1 \circledast \varphi_2$ states that the heap can be split in two disjoint (not necessarily contiguous) sub-heaps, and φ_1 holds in one and φ_2 in the other. The semantics of separation logic is given in terms of both an environment (or store, or a stack) and a *heap*. Separation logic program reasoning is still based on Hoare triples $\{\varphi\}\,\texttt{PGM}\,\{\varphi'\}$, but φ and φ' use separation logic.

We here present *matching logic*, which does not require, nor precludes, extensions of FOL. Instead, it proposes to encode program states as *(configuration) patterns*, which may, and typically do, contain variables and formulae among other items. The role of the variables is to allow patterns to be *matched* by concrete program configurations, and the role of the formulae is to constrain the matchings.

We discuss matching logic by defining and reasoning about KernelC, a fragment of C with `malloc` and `free`. For simplicity, we assume arbitrarily large (integer) numbers, infinite memory, and memory locations holding precisely one number. One can write many interesting C programs in KernelC. Here are some that we will refer to in the paper:

`SUM:` calculates in `s` the sum of the first `p` natural numbers:

```
s = 0;
n = 1;
while (n != p+1) {
  s = s+n;
  n = n+1;
}
```

`ALLOCATE:` allocates a single-linked list of 5 nodes, in reversed order, each node having two contiguous locations, one holding a value and the other a pointer to the next node:

```
n = 0;
p = null;
while (n != 5) {
  q = malloc(2);
  *q = n;
  *(q+1) = p;
  p = q;
  n = n+1;
}
```

`REVERSE:` reverses a list of nodes as above that starts at `p`:

```
if (p != null) {
  x = *(p+1);
  *(p+1) = null;
  while (x != null) {
    y = *(x+1);
    *(x+1) = p;
    p = x;
    x = y;
  }
}
```

`DEALLOCATE:` frees a list starting with `p`:

```
while (p != null) {
  q = *(p+1);
  free(p);
  p = q;
}
```

Matching logic is inspired from recent efforts in rewriting logic semantics (RLS) [12, 19, 16] of programming languages. Even though RLS definitions are apparently operational, they are, in fact, tightly connected to their matching logic equivalent formal systems. This connection has two major benefits: (1) it allows to prove *soundness* of matching logic formal systems w.r.t. executable and supposedly well-tested language semantics, and (2) it allows to *derive matching logic program verifiers* from executable RLS of languages. Moreover, the program verifiers resulting from (2) are themselves executable using rewriting logic.

We have implemented such a prover for KernelC using Maude [3] and experimented with it on tens of examples. For example, to prove the correctness of the last three programs above, notoriously difficult using conventional Hoare logics [15] and provers based on it, our matching logic prover requires minimal support from the user. The most difficult task is to equationally define an operator `list(p,A)` identifying, or matching, in the heap a flattened list structure as in C starting with pointer `p` and containing the sequence of integers `A`; as seen later in the paper, even this task is relatively easy. In experiments with our matching logic prover, we defined many other configuration constructs like `list`, including trees, queues, stacks, graphs, as well as parametric variants of them, e.g., stacks of trees, etc., and used them to verify several non-trivial programs, including the Schorr-Waite algorithm with arbitrary (cyclic) graphs.

In this paper we only focus on the mathematical foundations of matching logic: what it is and why it is sound. In passing, we also discuss memory safety issues and sketch our matching logic verification approach. Since matching logic is different from other logics for program verification, we start with a general overview of our approach in Section 2. Section 3 discusses rewriting logic semantics (RLS) and Section 4 gives an RLS to KernelC, defining also memory and strong memory safety. Section 5 presents matching logic in detail, proving its soundness for KernelC and discussing some differences and relationships to relevant related work. Section 6 shows how sound and complete (w.r.t. the matching logic formal system) program verifiers can be derived from K semantics definitions. Appendix 7 includes our current Maude implementation of a matching logic verifier using the approach in Section 6, together with many examples.

2. Overview of the Matching Logic Approach

Matching logic builds on top of rewriting logic semantics (RLS) [12, 19, 16]. A RLS consists of a set of equations and rewrite rules gradually rewriting *concrete configurations*, called for simplicity just *configurations*. A configuration in KernelC is a heterogeneous bag, called *cell*, containing 4 indexed items, also called subcells, of the form

$$\langle\langle K\rangle_k\ \langle\rho\rangle_{env}\ \langle\sigma\rangle_{mem}\ \langle\pi\rangle_{ptr}\rangle,$$

where: $\langle K\rangle_k$ is a fragment of code (K is a special list structure called *computation*), $\langle\rho\rangle_{env}$ is an environment map, $\langle\sigma\rangle_{mem}$ is a memory or heap allocation map, and $\langle\pi\rangle_{ptr}$ is a map storing for each allocated pointer the size of the allocated block (in C `free(P)` deallocates as many locations as previously allocated at `P` using an explicit `malloc`, so we need to keep this piece of information in the configuration for the semantics of `free`). The concrete configuration

$$\langle\langle \texttt{x=*y;y=x;}\rangle_k\ \langle \texttt{x}\mapsto 7,\texttt{y}\mapsto 6\rangle_{env}\ \langle 5\mapsto 3 \circledast 6\mapsto 4\rangle_{mem}\ \langle 5\mapsto 2\rangle_{ptr}\rangle$$

contains the code "`x=*y;y=x;`" in an environment mapping `x` to 7 and `y` to 6, a heap having value 3 at location 5 and 4 at 6, and where some previous `malloc` allocated 2 memory locations starting with location 5. Note that, unlike in separation logic where $\circledast$ is a logical connective, our $\circledast$ is a construct for maps represented as an algebraic data-type of sets of pairs $P \mapsto I$; hence, our $\circledast$ is a binary operation obeying laws like associativity, commutativity, etc. Different cells in the configuration may use different constructs (space, comma, etc.) as separators; we use $\circledast$ for memory as it resembles the separation conjunct in separation logic. KernelC's RLS rewrites the above configuration to (3 steps)

$$\langle\langle \texttt{y=x;}\rangle_k \, \langle \texttt{x}{\mapsto}4{,}\texttt{y}{\mapsto}6\rangle_{env} \, \langle 5{\mapsto}3 \circledast 6{\mapsto}4\rangle_{mem} \, \langle 5{\mapsto}2\rangle_{ptr}\rangle.$$

To define a matching logic formal system, one needs to first define the *configuration patterns*, called for simplicity just *patterns*. Patterns add variables and constraints over them to (concrete) configurations, thus allowing them to symbolically *specify* sets of configurations, namely all those *matching* the pattern consistently with its constraints. Variables can appear anywhere in the pattern: in computation, in environment, in memory, etc. Some of the variables appearing in a pattern, possibly all, can be *bound* by the pattern to localize their scope. The remaining unbound variables are called *free* for that pattern.

We formalize patterns by adding two more cells to configurations, one for its bound variables and another for its constraints. Constraints can be expressed using FOL, or fragments or extensions of it. Patterns used in our matching logic formal system of KernelC are bag cells containing 6 indexed sub-cells of the form

$$\langle\langle K\rangle_k \, \langle\rho\rangle_{env} \, \langle\sigma\rangle_{mem} \, \langle\pi\rangle_{ptr} \, \langle V\rangle_{bnd} \, \langle\varphi\rangle_{form}\rangle,$$

where the first 4 subcells are like in configurations but potentially using variables, and where $\langle V\rangle_{bnd}$ is a bag containing the pattern's bound variables and $\langle\varphi\rangle_{form}$ is a formula constraining the variables (both the bound and free ones).

A concrete program configuration γ *matches* configuration pattern $\Gamma = \langle\langle V\rangle_{bnd} \, \langle\varphi\rangle_{form} \, C\rangle$, written $\gamma \models \Gamma$, iff there is a *matching substitution* τ of the variables in Γ such that $\gamma = \langle\overline{\tau}(C)\rangle$ and $\overline{\tau}(\varphi)$ holds; if τ is important, then we write it as a subscript to $\models$, e.g., $\gamma \models_\tau \Gamma$. For instance, the pattern

$$\langle\langle \texttt{x=*y;}\rangle_k \, \langle \texttt{x}{\mapsto}p{,}\texttt{y}{\mapsto}p\rangle_{env} \, \langle p{\mapsto}q \circledast \sigma\rangle_{mem} \, \langle p\rangle_{bnd} \, \langle p{\neq}q\rangle_{form} \, C\rangle$$

is matched by a concrete configuration γ iff γ's current command is the assignment "`x=*y;`", where `x` and `y` are the only variables in the environment and are aliased to a pointer p holding some value, q, different from p; p is bound, so its scope is limited to the pattern, but q, σ and C are free. The difference between bound and free variables becomes relevant in the presence of correctness pairs, discussed next.

A matching logic formal system derives *correctness pairs* $\Gamma \Downarrow \Gamma'$, where Γ and Γ' are patterns. The intuition underlying correctness pairs $\Gamma \Downarrow \Gamma'$ is that they relate concrete program configurations with the resulting configurations after the enclosed fragment of program executes: if configuration γ yields configuration γ' after executing its enclosed code and if $\gamma \models_\tau \Gamma$, then $\gamma' \models_{\tau'} \Gamma'$ for some τ' with $\tau(x) = \tau'(x)$ for any variable x free in both Γ and Γ'. We keep the fragment of code embedded in the configurations instead of working with Hoare-like triples because expressions may have side effects which cannot be cleanly isolated (e.g., `malloc`). Dynamic logic [6] takes a similar approach.

Here is a correctness pair stating that `SUM` (see page 1) indeed calculates in `s` the sum of the first `p` natural numbers:

$$\begin{array}{l}\langle\langle \texttt{SUM}\rangle_k \, \langle \texttt{p} \mapsto p\rangle_{env} \, \langle p \geq 0\rangle_{form} \, \langle\cdot\rangle_{bnd} \, C\rangle \Downarrow \\ \langle\langle\cdot\rangle_k \, \langle \texttt{s} \mapsto p(p{+}1)/2, \rho\rangle_{env} \, \langle true\rangle_{form} \, \langle\rho\rangle_{bnd} \, C\rangle\end{array}$$

In words, the above says that if `SUM` is executed in a configuration in which `p` holds a value $p \geq 0$, then, in the resulting configuration ("$\cdot$" is the unit of any cell, including that of computations), `s` holds the value $p(p{+}1)/2$.

Such configuration pairs can be derived in matching logic using two types of rules, ones specific to each language definition, and a few general ones. These rules are discussed in detail in Section 5. Here we only discuss the matching logic rule for pointer assignment:

$$\frac{\langle\langle K_1\rangle_k C\rangle \Downarrow \langle\langle P\rangle_k C_1\rangle, \ \langle\langle K_2\rangle_k C_1\rangle \Downarrow \langle\langle I\rangle_k \, \langle P {\mapsto} I' \circledast \sigma\rangle_{mem} C_2\rangle}{\langle\langle {*}K_1{=}K_2\texttt{;}\rangle_k \, C\rangle \Downarrow \langle\langle\cdot\rangle_k \, \langle P {\mapsto} I \circledast \sigma\rangle_{mem} \, C_2\rangle}$$

In words, if a configuration embeds computation "$*K_1{=}K_2$;" with rest of configuration C, then first evaluate K_1 obtaining pointer P and configuration C_1 (if K_1 has side effects then $C \neq C_1$), then evaluate K_2 in C_1; if the result is I and the resulting configuration has pointer P allocated, then replace the value at that pointer by I keeping the rest of the configuration unchanged, and discard the assignment.

There are two other specific aspects of matching logic. One is *(configuration) pattern abstraction*, crucial for proof modularity: $\Gamma_1 \Rrightarrow \Gamma_2$ iff Γ_1 matches Γ_2 consistently with their internal constraints. Formally, if $\Gamma_1 = \langle\langle V_1\rangle_{bnd} \, \langle\varphi_1\rangle_{form} \, C_1\rangle$ and $\Gamma_2 = \langle\langle V_2\rangle_{bnd} \, \langle\varphi_2\rangle_{form} \, C_2\rangle$, then $\Gamma_1 \Rrightarrow \Gamma_2$ iff there is some substitution θ of the variables in V_2 such that $C_1 = \overline{\theta}(C_2)$ and $\varphi_1 \Rightarrow \overline{\theta}(\varphi_2)$ holds. Pattern abstraction captures the intuition that a state matching Γ_1 also matches Γ_2, so Γ_1 is "more concrete" and Γ_2 is "more abstract". Pattern abstraction is crucial for expressing loop invariants. For example, the following is a loop invariant pattern for the while loop in `SUM`:

$$\langle\langle \texttt{p} \mapsto p, \, \texttt{s} \mapsto n(n{-}1)/2, \texttt{n} \mapsto n\rangle_{env} \, \langle p{\geq}0 \wedge n{\leq}p{+}1\rangle_{form} \, \langle n\rangle_{bnd} \, C\rangle$$

Note that it binds n. After processing the body of the loop, the resulting pattern is an instance of this one with a substitution θ taking n to $n + 1$. Therefore, pattern abstraction is necessary in order to express and prove the loop invariant.

The other specific aspect of matching logic is the use of *(configuration) pattern equations*, written $t \equiv t'$ where t, t' are (subterms of) patterns, allowing terms to be replaced by equal terms in any pattern context. In other words, derivations in matching logic are done *modulo pattern equations*, the same way rewrites are done in rewriting logic modulo equations [11]. Pattern equational reasoning obeys all the conventional equational reasoning rules and assumes all the existing equations coming with the definition of configurations (associativity, commutativity, etc., of the cell constructs). Moreover, equalities implied by a pattern's constraints can also be used in equational derivations.

What makes pattern equational reasoning interesting is that one can add one's own pattern equational definitions. For example, one can define a heap construct *list* taking a pointer and a sequence of integers, together with two equations defining how and under what conditions it can be matched against a flat, pointer representation of a list in the heap. The two equations, together with pattern abstraction, allow us to identify sequences of integers as "mathematical objects" flattened in the configuration patterns. For example, one can show that

$$\langle\langle 3 \mapsto 1 \circledast 4 \mapsto 0 \circledast 5 \mapsto 2 \circledast 6 \mapsto 3\rangle_{mem}\, C\rangle \Rrightarrow \langle\langle list(5, 2.1)\rangle_{mem}\, C\rangle$$

for any well-formed C, where "_._" is the concatenation of sequences of integers and ϵ is the empty sequence, saying that the sequence of integers "2.1" can be identified in the heap as the elements of a list structure starting at pointer 5.

With the list construct for heaps, we can derive the following correctness pair for REVERSE (see page 1) using the matching logic formal system of KernelC in Section 5:

$$\begin{array}{l} \langle\langle \mathtt{REVERSE}\rangle_k\, \langle \mathtt{p} \mapsto p\rangle_{env}\, \langle list(p, \alpha)\rangle_{mem}\, \langle p\rangle_{bnd}\, \langle true\rangle_{form}\, C\rangle \\ \Downarrow \langle\langle\cdot\rangle_k \langle \mathtt{p} \mapsto p, \rho\rangle_{env} \langle list(p, rev(\alpha))\rangle_{mem} \langle p, \rho\rangle_{bnd} \langle true\rangle_{form} C\rangle. \end{array}$$

$rev(\alpha)$ is the reverse of α, an operation which is easy to define. Figure 6 depicts the detailed derivation; we only show the loop invariant pattern here (note that only α is free):

$$\left\langle \begin{array}{l} \langle \mathtt{p} \mapsto q, \mathtt{x} \mapsto r, \rho\rangle_{env}\, \langle list(q,\beta) \circledast list(r,\gamma)\rangle_{mem} \\ \langle rev(\alpha) = rev(\gamma).\beta\rangle_{form}\, \langle q,r,\rho,\beta,\gamma\rangle_{bnd} \end{array} \right\rangle$$

All the difficult tasks in matching logic uniformly rely on one principle, applied at various levels: *matching*. Matching (modulo associativity and commutativity of $\circledast$) also gives us *separation*: e.g., $\langle list(q,\beta) \circledast list(r,\gamma) \circledast \sigma\rangle_{mem}$ can only match three disjoint heaps: two lists and the rest. Even the difficult *frame inference* problem becomes a matching problem: e.g., in the rule for pointer assignment above, the (meta-)variables σ and C can be thought of as memory and configuration frames, respectively; to apply that rule (using the Substitution rule in Section 5), one needs to match its pre-configuration against the current configuration.

Unfortunately, matching is not an easy problem. However, like SAT, matching is relatively well understood and efficiently implemented by several systems; encouragingly, rewrite engines like ASF+SDF [20], Elan [2] and Maude [3] can execute millions of matching steps per second.

Matching logic is therefore a novel foundation for program verification, leveraging the strength of decision procedures for matching by uniformly recasting important program verification concepts as instances of matching.

3. Background: Rewriting Logic Semantics

Meseguer's rewriting logic [11] extends equational logic with rewrite rules. A rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, where Σ is a signature (sorts and operation symbols), E is a set of Σ-equations (written $t_1 = t_2$, where t_1, t_2 are Σ-terms possibly containing variables), and R is a set of Σ-rules (written $t_1 \to t_2$). Like in equational logic, terms can be replaced by equal terms in any context and in any direction. We write $\mathcal{R} \models t = t'$ whenever t can be proved equal to t' using equational deduction with the equations in $\mathcal{R}$. Like in term rewriting, rules can be applied in any context, but only from left-to-right. One way to think of rewriting logic is that equations apply until the term is *matched* by the left-hand-side (lhs) of some rule, which then irreversibly transforms the term. We write $\mathcal{R} \models t \to t'$ when t can be rewritten, using arbitrarily many equational steps but only one rewrite step in $\mathcal{R}$, into t'. Also, we write $\mathcal{R} \models t \to^* t'$ when t can be rewritten, using the equations and rules in $\mathcal{R}$, into t'. Rewriting logic thus captures *rewriting modulo equations* into a logic, with good mathematical properties (loose and initial models, complete deduction, proofs = computations, etc.). It is simple to understand and efficiently executable.

Rewriting logic semantics in general and the K technique in particular [12, 19, 16], referred to as "K" from here on, propose to define languages $\mathcal{L}$ as rewrite theories $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$, where $\Sigma_{\mathcal{L}}$ extends the syntax of $\mathcal{L}$. The equations $E_{\mathcal{L}}$ are thought of as structural rearrangements preparing the context for rules and carrying no computational meaning, while rules in $R_{\mathcal{L}}$ are irreversible computational steps. K achieves context-sensitivity in two ways: (1) by adding algebraic structure to configurations and using it to control matching; and (2) by extending the original language syntax with a special task sequentialization construct, "$\curvearrowright$" pronounced "then", as well as frozen variants of existing language constructs. Frozen operators have a "$\square$" as part of their name and are used to "freeze" fragments of program until their turn comes. Figure 1 shows the K semantics of a trivial assignment language, and a rewrite derivation in it.

In Figure 1, op stands for the various arithmetic and relational operations that one may want to include in one's language, and op_{Int} stands for the mathematical counterpart (function or relation) of op which operates on integers. For example, op can range over standard arithmetic operator names +, -, *, /, etc., and over standard relational oper-

$$
\begin{array}{lcll}
Int & ::= & \text{integer numbers} & \text{(abstract syntax)}\\
Id & ::= & \text{identifiers, to be used as variable names} & \\
K & ::= & Int \mid Id \mid K_1 \ \mathsf{op}\ K_2 \mid Id\texttt{=}K\texttt{;} \mid K_1\ K_2 &
\end{array}
$$

$$
\begin{array}{lcll}
Cfg & ::= & \langle \mathsf{Bag}_{\cdot}^{\_\,\_}[CfgItem]\rangle & \text{(configuration)}\\
CfgItem & ::= & \langle K\rangle_k \mid \langle Env\rangle_{env} & \\
K & ::= & \ldots \mid \mathsf{Seq}_{\cdot}^{\_\curvearrowright\_}[K] & \\
Env & ::= & \mathsf{Map}_{\cdot}^{\_,\_}[Id, Int] &
\end{array}
$$

$$
\begin{array}{ll}
K_1\ \mathsf{op}\ K_2 = (K_1 \curvearrowright \square\ \mathsf{op}\ K_2) & \text{(computation structural equations)}\\
I_1\ \mathsf{op}\ K_2 = (K_2 \curvearrowright I_1\ \mathsf{op}\ \square) & \\
(X\texttt{=}K\texttt{;}) = (K \curvearrowright X\texttt{=}\square\texttt{;}) &
\end{array}
$$

$$
\begin{array}{ll}
K_1\ K_2 = K_1 \curvearrowright K_2 & \text{(semantic equations and rules)}\\
I_1\ \mathsf{op}\ I_2 \rightarrow I_1\ op_{Int}\ I_2 & \\
\langle X \curvearrowright K\rangle_k\ \langle X \mapsto I, \rho\rangle_{env} \rightarrow \langle I \curvearrowright K\rangle_k\ \langle X \mapsto I, \rho\rangle_{env} & \\
\langle X\texttt{=}I\texttt{;} \curvearrowright K\rangle_k\ \langle \rho\rangle_{env} \rightarrow \langle K\rangle_k\ \langle \rho[X \leftarrow I]\rangle_{env} &
\end{array}
$$

(range of variables: $X \in Id$; $K, K_1, K_2 \in K$; $I, I_1, I_2 \in Int$)

$$
\begin{array}{lcl}
\langle\langle \texttt{x=7;y=x+3;x+y}\rangle_k\ \langle\cdot\rangle_{env}\rangle & =^* & \langle\langle \texttt{x=7;} \curvearrowright \texttt{y=x+3;} \curvearrowright \texttt{x+y}\rangle_k\ \langle\cdot\rangle_{env}\rangle \rightarrow\\
\langle\langle \texttt{y=x+3;} \curvearrowright \texttt{x+y}\rangle_k\ \langle \texttt{x}\mapsto 7\rangle_{env}\rangle & =^* & \langle\langle \texttt{x} \curvearrowright \square\texttt{+3} \curvearrowright \texttt{y=}\square\texttt{;} \curvearrowright \texttt{x+y}\rangle_k\ \langle \texttt{x}\mapsto 7\rangle_{env}\rangle \rightarrow\\
\langle\langle \texttt{7} \curvearrowright \square\texttt{+3} \curvearrowright \texttt{y=}\square\texttt{;} \curvearrowright \texttt{x+y}\rangle_k\ \langle \texttt{x}\mapsto 7\rangle_{env}\rangle & =^* & \langle\langle \texttt{y=7+3;} \curvearrowright \texttt{x+y}\rangle_k\ \langle \texttt{x}\mapsto 7\rangle_{env}\rangle \rightarrow\\
\langle\langle \texttt{y=10;} \curvearrowright \texttt{x+y}\rangle_k\ \langle \texttt{x} \mapsto 7\rangle_{env}\rangle & \rightarrow & \langle\langle \texttt{x+y}\rangle_k\ \langle \texttt{x} \mapsto 7,\ \texttt{y} \mapsto 10\rangle_{env}\rangle =\\
\langle\langle \texttt{x} \curvearrowright \square\texttt{+y}\rangle_k\ \langle \texttt{x}\mapsto 7,\ \texttt{y}\mapsto 10\rangle_{env}\rangle & \rightarrow & \langle\langle \texttt{7} \curvearrowright \square\texttt{+y}\rangle_k\ \langle \texttt{x}\mapsto 7,\ \texttt{y}\mapsto 10\rangle_{env}\rangle =^*\\
\langle\langle \texttt{y} \curvearrowright \texttt{7+}\square\rangle_k\ \langle \texttt{x}\mapsto 7,\ \texttt{y}\mapsto 10\rangle_{env}\rangle & \rightarrow & \langle\langle \texttt{10} \curvearrowright \texttt{7+}\square\rangle_k\ \langle \texttt{x}\mapsto 7,\ \texttt{y}\mapsto 10\rangle_{env}\rangle =\\
\langle\langle \texttt{7+10}\rangle_k\ \langle \texttt{x}\mapsto 7,\ \texttt{y}\mapsto 10\rangle_{env}\rangle & \rightarrow & \langle\langle \texttt{17}\rangle_k\ \langle \texttt{x}\mapsto 7,\ \texttt{y}\mapsto 10\rangle_{env}\rangle
\end{array}
$$

Figure 1. K semantics of simple assignment language and rewrite derivation (7 rewrites).

ator names ==, !=, <=, >=, etc., in which case $+_{Int}$ is the addition operation on integers (e.g., $3 +_{Int} 7 = 10$), etc., and $==_{Int}$ is the equality on integers (e.g., $(3 ==_{Int} 5) = 0$ and $(3 ==_{Int} 3) = 1$; for simplicity we assume, like in C, that boolean values are special integer values). One should add one group of 2 equations and 1 rule for each such arithmetic or relational operator name that one wants in the language, as we generically showed in Figure 1.

K definitions typically use only one (abstract) syntactic category, K, serving as minimal syntactic infrastructure to define terms; it is not intended to be used for parsing or type-checking. We make no distinction between algebraic signatures and their context-free notation: syntactic categories correspond to sorts and productions to operations in the signature; for example, production "K ::= $Id\texttt{=}K\texttt{;}$" is equivalent to defining an operation "$\_\texttt{=}\_\texttt{;} : Id \times K \rightarrow K$".

Sequences and bags are standard (equational) datastructures. We use notations $\mathsf{Seq}_u^{\_@\_}[S]$ for sequences and $\mathsf{Bag}_u^{\_@\_}[S]$ for bags, resp., where u is their unit and $\_@\_$ is their binary construct. Formally, if added for sort S', these correspond to adding subsorting $S < S'$ (i.e., production S' ::= S, not needed when $S' = S$), operations $u :\rightarrow S'$ (a constant) and $\_@\_ : S' \times S' \rightarrow S'$, and appropriate unit and associativity equations for sequences, and unit, commutativity and associativity equations for bags; e.g., the third production in box (configuration) in Figure 1 desugars as:

$$
\begin{array}{lr}
K ::= \ldots \mid \cdot \mid K \curvearrowright K & \text{// additional sequence constructs}\\
(\cdot \curvearrowright K) = (K \curvearrowright \cdot) = K & \text{// unit}\\
(K_1 \curvearrowright K_2) \curvearrowright K_3 = K_1 \curvearrowright (K_2 \curvearrowright K_3) & \text{// associativity}
\end{array}
$$

We also assume finite maps; formally, $\mathsf{Map}_u^{\_@\_}[S_1, S_2]$ corresponds to bags of pairs of elements of sorts S_1 and S_2, respectively, each pair written $s_1 \mapsto s_2$, with additional operations $\_[\_] : S' \times S_1 \rightarrow S_2$ and $\_[\_ \leftarrow \_] : S' \times S_1 \times S_2 \rightarrow S'$ and $\_\backslash\_ : S' \times S_1 \rightarrow S'$ for lookup, update (adding a new pair if map undefined on that element) and deletion (i.e., removing an element binding), respectively, where S' is the sort corresponding to the maps. These operators are overloaded. E.g., "Env ::= $\mathsf{Map}_{\cdot}^{\_,\_}[Id, Int]$" in Figure 1 desugars as:

$$
\begin{array}{lr}
Env ::= \cdot \mid Id \mapsto Int \mid Env, Env \mid Env[Id \leftarrow Int] \mid Env\backslash Id & \\
Int ::= \ldots \mid Env[Id] & \\
\rho\ ,\ \cdot = \cdot\ ,\ \rho = \rho & \text{// unit}\\
\rho_1\ ,\ \rho_2 = \rho_2\ ,\ \rho_1 & \text{// commutativity}\\
\rho_1\ ,\ (\rho_2\ ,\ \rho_3) = (\rho_1\ ,\ \rho_2)\ ,\ \rho_3 & \text{// associativity}\\
(\rho\ ,\ (X \mapsto I))[X] = I & \\
(\rho\ ,\ (X \mapsto I))[Y] = \rho[X] & \text{when } X \neq Y\\
\ldots \text{ the remaining equations are similar } \ldots &
\end{array}
$$

Hence, an environment is a finite bag of pairs, $\rho[X]$ retrieves the Int associated to the Id X in ρ, $\rho[X \leftarrow J]$ updates the Int corresponding to X in ρ to J, and $\rho\backslash X$ removes pair $X \mapsto \_$ from ρ (if there is any). One can also define, in the same style, an operation Dom giving the domain of a map as a bag of elements, as well as an operation checking whether the map term is indeed a partial function. These operations are easy to define algebraically and therefore we assume them from here on; in fact, we assume that each map that occurs in an equation or rule is a well-formed map (e.g., the maps $\sigma \circledast \sigma'$ in the rules for `malloc` and `free` in Figure 2).

Sequences, bags and maps are core to K language definitions. When used as configuration constructors, we call them *K-cells* or just *cells*. K is a *modular* definitional framework: rules match only what they need from the configuration, so one can change the configuration (e.g., adding store, input/output, stacks, etc.) without having to revisit existing rules. In particular, the KernelC semantics (Fig. 2) includes that of the simple language in Fig. 1 *unchanged*, despite additional K-cells in the configuration of KernelC. Matching logics with K configurations inherit the modularity of K.

Sort K contains *computation structures*, or simply *computations*, obtained by adding to the original abstract syntax *computation sequences* (terms in $\mathsf{Seq}_{\cdot}^{\_\curvearrowright\_}[K]$) and *frozen computations* (wrapped by operators containing a "$\square$" in their name). Intuitively, $K_1 \curvearrowright K_2$ means "first process K_1, *then* process K_2". Frozen computations are structurally inhibited from advancing until their turn comes. For example, "$K_1\ \mathsf{op}\ K_2$" first processes K_1 and in the meanwhile keeps K_2

frozen: "K_1 op K_2 = $K_1 \curvearrowright \square$ op K_2". After K_1 is processed, its result is placed back in context and K_2 is "scheduled": "I_1 op $K_2 = K_2 \curvearrowright I_1$ op $\square$". As equations, these can be applied forth (to "schedule" for processing) and back (to "plug" results back). We assume all freezing operators are automatically added to sort K (i.e., the "..." in "K ::= ..." in Figure 1 include "Id=$\square$;" and "$\square$ op K | K op $\square$" for all operations op that one in the language). Computation equations give the evaluation strategy of each language construct. They accomplish the same role as the context productions of evaluation contexts [22], but logically rather than syntactically.

We next discuss the semantic equations and rules in Figure 1. When the language has sequential composition, it is typically desugared into K's $\curvearrowright$. The rewrite semantics of + is clear. The first rule making use of the structure of the configuration is the variable lookup rule: it matches the top of the $\langle ... \rangle_k$ cell, which must be a variable identifier $X \in Id$, as well as a pair $X \mapsto I$ in the environment cell, and replaces X in the computation by I. The rule for variable assignment updates the environment, at the same time dissolving the assignment statement. We chose to let lookup of uninitialized variables be undefined; thus, the term $\langle\langle \texttt{x=x+1;}\rangle_k \langle\cdot\rangle_{env}\rangle$, which is equal to $\langle\langle \texttt{x} \curvearrowright \square\texttt{+1} \curvearrowright \texttt{x=}\square\texttt{;}\rangle_k \langle\cdot\rangle_{env}\rangle$, is stuck.

K relates to reduction semantics (with [22] and without [13] evaluation contexts), SECD [9] and other abstract machines, the CHAM [1], continuations [14], refocusing [4], etc. We refer the interested reader to [16] for details on K.

4. KernelC

We here discuss the K definition of KernelC together with some memory safety aspects.

4.1. Formal Semantics of KernelC

Figure 2 shows the complete K definition of KernelC, a C-like language with dynamic memory allocation and deallocation. We assume programs syntactically correct:

Definition 1 *A* KernelC *computation K is* ***well-formed*** *iff it is equal (using equational reasoning within* KernelC*'s semantics) to a well-formed program or expression in C. Also, a computation is* ***well-terminated*** *iff it is equal to the unit computation "$\cdot$" or to an integer value $I \in Int$.*

We also assume the C meaning of the language constructs. In particular, malloc(N) allocates a block of N contiguous locations and returns a pointer to the first location, and free(P) assumes that a block of N locations has been previously allocated using a corresponding malloc and deallocates all N locations. For simplicity, we allow integers of any size and assume that locations, which are addressed using natural numbers, can hold any integer.

Nat ::= naturals, Int ::= integers (abstract syntax)
Id ::= identifiers, to be used as variable names
K ::= Int | Id | null | *K | !K | K_1 op K_2 | K_1 && K_2 | K_1 || K_2 |
 | K_1=K_2; | K_1 K_2 | {K} | {} | malloc(K); | free(K);
 | if (K_1) K_2 | if (K_1) K_2 else K_3 | while (K_1) K_2

null = 0 (desugaring of non-core constructs)
! K = if (K) 0 else 1
K_1 && K_2 = if (K_1) K_2 else 0
K_1 || K_2 = if (K_1) 1 else K_2
{K} = K
if (K_1) K_2 = if (K_1) K_2 else {}

Cfg ::= $\langle \mathsf{Bag}^{\text{--}}_{\cdot}[CfgItem]\rangle$ (configuration)
$CfgItem$ ::= $\langle K\rangle_k \mid \langle Env\rangle_{env} \mid \langle Mem\rangle_{mem} \mid \langle Ptr\rangle_{ptr}$
K ::= ... | $\mathsf{Seq}^{\text{-}\curvearrowright\text{-}}_{\cdot}[K]$
Mem ::= $\mathsf{Map}^{\text{-}\circledast\text{-}}_{\cdot}[Nat^+, Int]$
Env ::= $\mathsf{Map}^{\text{-},\text{-}}_{\cdot}[Id, Int]$
Ptr ::= $\mathsf{Map}^{\text{-},\text{-}}_{\cdot}[Nat^+, Nat]$

*K = ($K \curvearrowright$ *$\square$) (computation structural equations)
K_1 op K_2 = ($K_1 \curvearrowright \square$ op K_2)
I_1 op K_2 = ($K_2 \curvearrowright I_1$ op $\square$)
if (K_1) K_2 else K_3 = ($K_1 \curvearrowright$ if ($\square$) K_2 else K_3)
(X=K;) = ($K \curvearrowright X$=$\square$;)
(*K_1=K_2;) = ($K_1 \curvearrowright$ *$\square$=K_2;)
(*P_1=K_2;) = ($K_2 \curvearrowright$ *P_1=$\square$;)
malloc(K); = ($K \curvearrowright$ malloc($\square$);)
free(K); = ($K \curvearrowright$ free($\square$);)

{} = $\cdot$ (semantic equations and rules)
$K_1\ K_2 = K_1 \curvearrowright K_2$
I_1 op $I_2 \rightarrow I_1\ op_{Int}\ I_2$
if (0) K_2 else $K_3 \rightarrow K_3$
if (I) K_2 else $K_3 \rightarrow K_2$ where $I \neq 0$
$\langle X \curvearrowright K\rangle_k \langle X \mapsto I, \rho\rangle_{env} \rightarrow \langle I \curvearrowright K\rangle_k \langle X \mapsto I, \rho\rangle_{env}$
$\langle X\texttt{=}I\texttt{;} \curvearrowright K\rangle_k \langle \rho\rangle_{env} \rightarrow \langle K\rangle_k \langle \rho[X \leftarrow I]\rangle_{env}$
$\langle {*}P \curvearrowright K\rangle_k \langle P \mapsto I \circledast \sigma\rangle_{mem} \rightarrow \langle I \curvearrowright K\rangle_k \langle P \mapsto I \circledast \sigma\rangle_{mem}$
$\langle {*}P\texttt{=}I\texttt{;} \curvearrowright K\rangle_k \langle P \mapsto I' \circledast \sigma\rangle_{mem} \rightarrow \langle K\rangle_k \langle P \mapsto I \circledast \sigma\rangle_{mem}$
$\langle \texttt{while(}K_1\texttt{)}K_2 \curvearrowright K\rangle_k = \langle \texttt{if(}K_1\texttt{)\{}K_2\texttt{;while(}K_1\texttt{)}K_2\texttt{\}} \curvearrowright K\rangle_k$
$\langle \texttt{malloc(}N\texttt{);} \curvearrowright K\rangle_k \langle \sigma\rangle_{mem} \langle \pi\rangle_{ptr} \rightarrow \langle P \curvearrowright K\rangle_k \langle \sigma \circledast \sigma'\rangle_{mem} \langle \pi[P \leftarrow N]\rangle_{ptr}$
where $Dom(\sigma') = \overline{P, P+N-1}$
$\langle \texttt{free(}P\texttt{);} \curvearrowright K\rangle_k \langle \sigma \circledast \sigma'\rangle_{mem} \langle P \mapsto N, \pi\rangle_{ptr} \rightarrow \langle K\rangle_k \langle \sigma\rangle_{mem} \langle \pi\rangle_{ptr}$
where $Dom(\sigma') = \overline{P, P+N-1}$

(range of variables: $X \in Id$; $K, K_1, K_2 \in K$; $I, I_1, I_2 \in Int$; $P \in Nat^+$; $N \in Nat$)

Figure 2. KernelC **in K: Complete Semantics**

The desugaring equations are self-contained (Figure 2); we prefer to desugar derived language constructs wherever possible. The "boolean" constructs && and || are shortcut. Even though the conditional is a statement, once all syntactic categories are collapsed into one, K, it can be used to desugar expression constructs as well. We use parentheses for grouping, as in the equation desugaring `!=`.

The configuration of KernelC is a top $\langle ... \rangle$ cell containing four sub-cells: the $\langle ... \rangle_k$ and $\langle ... \rangle_{env}$ cells also present in the simple language in Figure 1; a cell $\langle ... \rangle_{mem}$ holding the memory (or heap) which can be dynamically allocated/deallocated; and a cell $\langle ... \rangle_{ptr}$ associating to pointers returned by `malloc` the number of locations that have been allocated (this info is necessary for the semantics of `free`).

Definition 2 *Let (Σ, E) be the algebraic specification of* KernelC *configurations: Σ contains all the configuration constructs (for bags, maps, etc.) and E contains all their defining equations (associativities, commutativities, etc.). Let $\mathcal{T}$ be the Σ-algebra of ground terms; the E-equational classes (i.e., provably equal using equational reasoning with E) of (ground) terms in $\mathcal{T}$ of sort Cfg which have the form $\langle\langle K\rangle_k\, \langle\rho\rangle_{env}\, \langle\sigma\rangle_{mem}\, \langle\pi\rangle_{ptr}\rangle$ are called **(concrete) configurations**. We distinguish several types of configurations:*

- *Configurations of the form $\langle\langle K\rangle_k\, \langle\cdot\rangle_{env}\, \langle\cdot\rangle_{mem}\, \langle\cdot\rangle_{ptr}\rangle$ where K is a well-formed computation, also written more compactly $[\![K]\!]$, are called **initial configurations**;*
- *Configurations $\langle\langle K\rangle_k\, \langle\rho\rangle_{env}\, \langle\sigma\rangle_{mem}\, \langle\pi\rangle_{ptr}\rangle$ whose embedded computation K is well-terminated (a "$\cdot$" or an $I \in Int$) are called **final configurations**;*
- *Configurations $\gamma \in \mathcal{T}$ which cannot be rewritten anymore (i.e., there is no configuration $\gamma' \in \mathcal{T}$ such that* KernelC$\models \gamma \to \gamma'$*) are **normal form configurations**;*
- *Normal form configurations which are not final are called **stuck (or junk, or core dump) configurations**;*
- *Configurations γ which cannot be rewritten infinitely (i.e., there is no infinite set of configurations $\{\gamma_n\}_{n\in Nat}$ such that $\gamma_0 = \gamma$ and* KernelC $\models \gamma_n \to \gamma_{n+1}$ *for any $n \in Nat$) are called **terminating configurations**.*

The computation structural equations define the desired evaluation strategy of each of the language constructs. Note the one for the conditional, which schedules for processing the condition, keeping the two branches frozen.

Let us discuss the semantic equations and rules in Figure 2. Empty blocks and sequential composition are dissolved into the unit and the sequentialization of K. The rules for +, == and `if` are clear. Variable lookup and assignment rules are the same as in Figure 1. Pointer lookup and update are similar, replacing the environment by memory, with a subtle difference: the rule for pointer assignment matches the pointer in the memory map, so it correctly requires the pointer to be already allocated in memory; for environments, we preferred to add an environment entry in case the variable is not present in the environment (that was because we do not have explicit variable declarations in KernelC; if we had, than the rules for variable and pointer lookup would be similar, the former resembling the latter).

The equation of `while` shows a use of the cell structure to achieve context sensitivity; replacing it with the simple-minded equation (or rule in case one prefers to regard loops unrolling as a computational step)

$$\texttt{while(}K_1\texttt{)}K_2 = \texttt{if(}K_1\texttt{)\{}K_2\texttt{;while(}K_1\texttt{)}K_2\texttt{\}}$$

then there is nothing to prevent the application of this equation again on the while term inside the conditional, and so on. While proof-theoretically one could argue that there is no problem with that, operationally it is problematic as it leads to operational non-termination even though the program may terminate. The equation of `while` applies only when `while` is the first computation task; it cannot apply again until the outer conditional and K_2 are processed.

Figure 3 shows a rewriting logic derivation using the K semantics in Figure 2; $\to^*$ stands for one or more rewrite steps, with arbitrarily many equational steps in between.

The rules for `free` and `malloc` make subtle use of matching modulo associativity and commutativity of $\_\circledast\_$. In the case of `free(`P`)`, a σ' is matched in the $\langle ... \rangle_{mem}$ cell whose domain is the N contiguous locations $\overline{P, P+N-1}$, where N is the natural number associated to P in the $\langle ... \rangle_{ptr}$ cell (i.e., the number of locations previously allocated at P using a `malloc`); then the `free` statement in cell $\langle ... \rangle_k$, the memory map σ' in cell $\langle ... \rangle_{mem}$ and the pointer mapping $P\mapsto N$ in cell $\langle ... \rangle_{ptr}$ are discarded; this way, the memory starting with location P can be reclaimed and reused in possible implementations of KernelC. Recall that we assume that all (partial) maps appearing in any context are well-formed; in particular, the map $\sigma \circledast \sigma'$ in the rule of `free` is well-formed, which means that there is only one such matching in the memory cell (P and N are given), which means that the rule for `free` is deterministic. Such a compact and elegant definition is possible only thanks to the strength of matching and rewriting modulo equations. Maude [3] provides efficient support for these operations, which is what makes it a very convenient execution vehicle for K and matching logic. The well-formedness of maps can either be assumed (one can prove aside that each equation/rule preserves it) or checked as a condition attached to the rule.

The most intricate rule in Figure 2 is that of `malloc`, which is an almost exact dual of the rule for `free`. Like in the `free` rule, the σ' is doubly constrained: its domain is disjoint from σ's (because $\sigma \circledast \sigma'$ is well-formed) and its domain is the set of contiguous locations $\overline{P, P+N-1}$ with P the returned pointer. However, the constraints on σ' are loose enough to allow a high degree of semantic non-determinism. E.g., program "`BAD` $\equiv$ `p=malloc(2);*2=7;`" may exhibit three different types of behavior, two in which it

Let REVERSE be the list reverse program in Introduction, and let
$\texttt{WHILE} \equiv \texttt{while(x!=null)}\{\texttt{y=*(x+1);*(x+1)=p;p=x;x=y;}\}$
$\texttt{IF} \equiv \texttt{if}(\square)\{\texttt{x=*(p+1); *(p+1)=null; WHILE}\}$.
Also, let us assume the environment and memory maps:
$(\rho_1 \equiv p \mapsto 1, x \mapsto 0, y \mapsto 0)$, $(\rho_2 \equiv p \mapsto 1, x \mapsto 5, y \mapsto 0)$, $(\rho_3 \equiv p \mapsto 5, x \mapsto 0, y \mapsto 0)$, $(\sigma_1 \equiv 1 \mapsto 7 \circledast 2 \mapsto 5 \circledast 5 \mapsto 9 \circledast 6 \mapsto 0)$, $(\sigma_2 \equiv 1 \mapsto 7 \circledast 2 \mapsto 0 \circledast 5 \mapsto 9 \circledast 6 \mapsto 0)$, $(\sigma_3 = 1 \mapsto 7 \circledast 2 \mapsto 0 \circledast 5 \mapsto 9 \circledast 6 \mapsto 1)$. The following derivation shows an execution reversing a list with the elements 7, 9:

$$\begin{array}{ll}
\langle \texttt{REVERSE(p)} \rangle_k \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} = \langle \texttt{p!=null} \curvearrowright \texttt{IF} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} = \\
\langle \texttt{!(p==null)} \curvearrowright \texttt{IF} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} =^* \\
\langle \texttt{if (p==0) 0 else 1} \curvearrowright \texttt{IF} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} = \\
\langle \texttt{p==0} \curvearrowright \texttt{if} (\square) \texttt{ 0 else 1} \curvearrowright \texttt{IF} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} = \\
\langle \texttt{p} \curvearrowright \square\texttt{==0} \curvearrowright \texttt{if} (\square) \texttt{ 0 else 1} \curvearrowright \texttt{IF} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} \rightarrow \\
\langle 1 \curvearrowright \square\texttt{==0} \curvearrowright \texttt{if} (\square) \texttt{ 0 else 1} \curvearrowright \texttt{IF} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} = \\
\langle \texttt{1==0} \curvearrowright \texttt{if} (\square) \texttt{ 0 else 1} \curvearrowright \texttt{IF} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} \rightarrow \\
\langle 0 \curvearrowright \texttt{if} (\square) \texttt{ 0 else 1} \curvearrowright \texttt{IF} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} = \\
\langle \texttt{if (0) 0 else 1} \curvearrowright \texttt{IF} \rangle_k \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} \rightarrow \langle 1 \curvearrowright \texttt{IF} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} \rightarrow^* \\
\langle \texttt{x=*(p+1);} \curvearrowright \texttt{*(p+1)=0;} \curvearrowright \texttt{WHILE} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} \rightarrow^* \\
\langle \texttt{x=*2;} \curvearrowright \texttt{*(p+1)=0;} \curvearrowright \texttt{WHILE} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} \rightarrow^* \\
\langle \texttt{x=5;} \curvearrowright \texttt{*(p+1)=0;} \curvearrowright \texttt{WHILE} \rangle_k & \langle \rho_1 \rangle_{env} \langle \sigma_1 \rangle_{mem} \rightarrow^* \\
\langle \texttt{*(p+1)=0;} \curvearrowright \texttt{WHILE} \rangle_k & \langle \rho_2 \rangle_{env} \langle \sigma_1 \rangle_{mem} \rightarrow^* \\
\langle \texttt{if(x!=0)}\{\texttt{y=*(x+1);*(x+1)=p;p=x;x=y;WHILE}\} \rangle_k & \langle \rho_2 \rangle_{env} \langle \sigma_2 \rangle_{mem} \rightarrow^* \\
\langle \texttt{y=*(x+1);} \curvearrowright \texttt{*(x+1)=p;p=x;x=y;WHILE} \rangle_k & \langle \rho_2 \rangle_{env} \langle \sigma_2 \rangle_{mem} \rightarrow^* \\
\langle \texttt{*(x+1)=p;} \curvearrowright \texttt{p=x;} \curvearrowright \texttt{x=y;} \curvearrowright \texttt{WHILE} \rangle_k & \langle \rho_2 \rangle_{env} \langle \sigma_2 \rangle_{mem} \rightarrow^* \\
\langle \texttt{p=x;} \curvearrowright \texttt{x=y;} \curvearrowright \texttt{WHILE} \rangle_k & \langle \rho_2 \rangle_{env} \langle \sigma_3 \rangle_{mem} \rightarrow^* \\
\langle \texttt{if(x!=0)}\{\texttt{y=*(x+1);*(x+1)=p;p=x;x=y;WHILE}\} \rangle_k & \langle \rho_3 \rangle_{env} \langle \sigma_3 \rangle_{mem} \rightarrow^* \\
\langle \cdot \rangle_k \langle \rho_3 \rangle_{env} \langle \sigma_3 \rangle_{mem} &
\end{array}$$

Figure 3. Rewriting logic derivation using the KernelC **semantics in Figure 2.**

terminates normally but in non-isomorphic configurations, and one in which it gets stuck looking up for location 1 which is not allocated. E.g., $\langle \langle \texttt{BAD} \rangle_k \langle \cdot \rangle_{env} \langle \cdot \rangle_{mem} \langle \cdot \rangle_{ptr} \rangle$ rewrites to any of the following (each being a normal form):

$$\langle \langle \cdot \rangle_k \, \langle \texttt{p} \mapsto 1 \rangle_{env} \, \langle (1 \mapsto -1) \circledast (2 \mapsto 7) \rangle_{mem} \, \langle 1 \mapsto 2 \rangle_{ptr} \rangle$$
$$\langle \langle \cdot \rangle_k \, \langle \texttt{p} \mapsto 2 \rangle_{env} \, \langle (2 \mapsto 7) \circledast (3 \mapsto -1) \rangle_{mem} \, \langle 2 \mapsto 2 \rangle_{ptr} \rangle$$
$$\langle \langle \texttt{*}2 \curvearrowright \square\texttt{=7;} \rangle_k \langle \texttt{p} \mapsto 5 \rangle_{env} \langle (5 \mapsto -1) \circledast (6 \mapsto -3) \rangle_{mem} \langle 5 \mapsto 2 \rangle_{ptr} \rangle$$

In concrete implementations of KernelC, one may see the last type of behavior more frequently than the other two, as it is little likely that `malloc` allocates at the "predicted" location, 2 in our case. We tried this code in `gcc` on a Linux machine (casting 1 to `(T*)1`) and it compiled (but it gave an expected segmentation fault when run). Thus, we can regard the third normal form term above as a "core dump".

We claim that, in spite of this apparently undesired non-determinism, this is the most general semantics of `malloc` that a language designer may want to have[1]. Any other additional constraints, such as "always allocate a fresh memory region", or "always reuse existing memory if possible", etc., may lead to a restrictive definition of KernelC, possibly undesired by some implementors. The actual C language makes no specific requirements on memory allocation, allowing C interpreters or compilers freedom to choose among various memory allocation possibilities; it is programmers' responsibility to write programs that do not rely on particular memory allocation strategies.

We can now formally state what KernelC is:

Definition 3 *The language* KernelC *discussed in this section is the rewrite logic theory* $(\Sigma_{\text{KernelC}}, E_{\text{KernelC}}, R_{\text{KernelC}})$ *depicted in Figure 2. If* KernelC $\models \gamma \rightarrow^* \gamma'$ *we say that, in* KernelC*, configuration* γ ***rewrites to*** *configuration* γ'.

Both the abstract syntax of KernelC and Σ are included in Σ_{KernelC}, and also both the desugaring equations of derived KernelC constructs and E are included in E_{KernelC}; recall from Definition 2 that (Σ, E) is the equational definition of KernelC configurations.

Therefore, the rewrite logic semantics of KernelC, identified with KernelC from here on, can produce by means of rewriting all the possible complete or intermediate executions that the language can yield. In particular, if

$$\text{KernelC} \models [\![K]\!] \rightarrow^* \gamma$$

with K a well-formed computation and γ a final configuration, then γ contains the (possibly non-deterministic) "result" obtained after "evaluating" K. In addition to comprising all the good executions, the rewrite theory KernelC also comprises all the bad executions of KernelC programs, namely all those that can get stuck; as seen shortly, this is very important as it will allow us to formally define memory safety of KernelC programs.

Note that like in any other formal operational semantics, our rewrite logic definition of KernelC has the property that informal execution steps and whole executions of programs become, respectively, formal proof steps and whole proofs in rewriting logic. Interestingly, unlike in other operational semantic frameworks, rewriting logic also provides models which are complete for its proof system, so the very same K definition of KernelC is also a loose "denotational" semantics in addition to being an "operational" one; moreover, since rewriting logic admits initial models, which are essentially built as a fix point over the algebra of terms, there is a selected subset of models, the "reachable" ones, for which induction is valid. As seen in Section 5, one can also systematically obtain a matching logic proof system from the same K definition of KernelC, which can be used to formally prove properties about programs. In other words, once one has a K definition of a language, one needs no other formal semantics of that language because its K definition already provides everything one may need from a formal semantics. This is also one of the reasons for which we call K semantics *executable* rather than operational; the

[1]To accommodate some implementations, one may want to have an even more general definition of `malloc(N)`, namely one in which *at least* N locations are allocated; we do not do it here, but it can be easily done by replacing the second and third occurrences of N in the rule for `malloc` with an M and adding a side condition $M \geq N$ to the rule.

latter may give the wrong impression that the K semantics can only be used to yield an interpreter for the language.

Even though K is executable by its very nature, here we actually *defined*, and not *implemented*, KernelC. We therefore wanted to keep our semantics as loose, or unconstrained, as possible. As usual, when implementing non-deterministic specifications one needs not (and typically does not) provide all the non-deterministic behaviors in one's implementation. In fact, each implementation of KernelC is expected to be deterministic. The non-determinism of `malloc` in our KernelC definition is a result of a deliberate language *under-specification*, not a desired non-deterministic feature of the language. General details on under-specification versus non-determinism are beyond our scope here, but the interested reader is referred to [21] for an in-depth discussion on these subjects. An additional advantage of the under-specified `malloc` in our definition of KernelC is that it allows us to elegantly yet rigorously define memory safety in the next section: a program is memory-safe iff it cannot get stuck, i.e., it cannot be rewritten to a normal form whose computation cell is different from "$\cdot$" or an integer. It is worth mentioning here that the soundness theorem of matching logic (Theorem 27) also ensures the memory safety of the verified program.

4.2. Memory Safety and SafeKernelC

We here give a formal definition to *memory safety* in KernelC, capturing the intuition that a program is memory safe iff it is so under any possible implementation of KernelC, i.e., under any possible choice the rule for `malloc` may make. Due to the undecidability of termination in general, our notion of memory safety, like any other practical (i.e., not unreasonably restricted) notion of memory safety, is undecidable in general. In this section we show that memory safety is actually undecidable even for terminating KernelC programs. That means, in particular, that KernelC (as an executable semantics) as well as any faithful implementation of it, cannot detect memory safety violations even on programs which always terminate, no matter whether that is attempted statically or at runtime.

To check memory safety, one therefore needs either to rely on user help (e.g., annotations), which is our approach in Section 5 in a more general verification setting including memory safety, or to restrict the class of memory safe programs, which is what we do next. We propose the semantic notion of *strong memory safety*: a program is strongly memory safe iff it does not get stuck in the executable semantics SafeKernelC, a variant of KernelC semantics with symbolic pointers. Interestingly, our formal definition of strong memory safety includes the informal notion of memory safety implied by the "C rules for pointer operations" [5]. Strong memory safety is shown decidable for terminating programs, but, of course, it is undecidable in general.

Definition 4 *Well-formed computation K is* ***terminating*** *iff $[\![K]\!]$ is a terminating configuration in* KernelC, *and is* ***memory safe*** *iff any normal form of $[\![K]\!]$ in* KernelC *is final.*

Program "`BAD` $\equiv$ `p=malloc(2);*2=7;`" is terminating but not memory safe: $[\![\texttt{BAD}]\!]$ rewrites, as seen, to normal form $\langle\langle *2 \curvearrowright \square\texttt{=7;}\rangle_k \langle \texttt{p} \mapsto 5\rangle_{env} \langle (5 \mapsto -1) \circledast (6 \mapsto -3)\rangle_{mem} \langle 5 \mapsto 2\rangle_{ptr}\rangle$. Program "`GOOD` $\equiv$ `p=malloc(2);*(p+1)=7;`", on the other hand, is both terminating and memory-safe: $[\![\texttt{GOOD}]\!]$ rewrites only to normal form configurations of the form $\langle\langle\cdot\rangle_k \langle \texttt{p} \mapsto i\rangle_{env} \langle (i \mapsto j) \circledast (i+1 \mapsto 7)\rangle_{mem} \langle i \mapsto 2\rangle_{ptr}\rangle$, where $i \in Nat^+$ and $j \in Int$. Program "`p=malloc(1);while(*p){}`" is memory safe but not terminating (when `*p` $\neq 0$), and finally, program "`p=malloc(1);while(*1){}`" is neither memory-safe (when `p` $\neq 1$) nor terminating (when `p` $= 1$ and `*1` $\neq 0$).

For our simple language, memory is the only source of unsafety; for more complex languages, one may have various types of safety, depending upon the language construct at the top of the computation in t when t is a normal form, which tells why the computation got stuck; e.g., if the language has division and $3/0$ is at the top of the computation, then K got stuck because a division by zero was attempted.

KernelC is Turing complete (we assumed both arbitrarily large integers and infinite memory), so termination of KernelC programs is undecidable. That immediately implies that memory safety is also undecidable in general: for any memory safe program `PGM`, the program "`PGM;BAD`" is memory safe iff `PGM` does not terminate. What is not so obvious is the decidability or undecidability of memory safety on terminating programs. In the remaining of this section we show that this is actually an undecidable problem, but that a stronger version of memory safety is decidable.

A hasty reader may think that, since programs have no symbolic inputs or data, memory safety must be decidable on terminating programs: one can simply run the program and check each memory access. The complexity of the problem comes from the non-determinism/under-specification of `malloc`, which makes any particular execution of the program to mean close to nothing wrt memory safety. Consider, for example, an execution of the program "`x=malloc(1); free(x); y=malloc(1); *x=1;`" in which the second `malloc` *just happens* to return the same pointer as the first `malloc`. Since this particular execution taking place on a hypothetical particular implementation of KernelC terminates normally, one may be wrongly tempted to say that it is memory safe; this program is clearly *not* memory safe (gets stuck if second `malloc` chooses a different location) and even the execution itself can be argued as memory unsafe, because of a memory leak on `x` (dangling pointer).

Since unrestricted use of pointers returned by `malloc` can lead to non-deterministic executions of programs, one could, in principle, introduce some notion of "path memory safety". For example, one could argue that an execution of the program "`x=malloc(1); y=malloc(1); if (y==x+1) {} else BAD`" in which `y` just happens to be `x+1` is memory safe, or that an execution of the program "`x=malloc(2); if (*x==*(x+1)) {} else BAD`" in which `*x` just happens to be `*(x+1)` is memory safe. Encouraged by the informal so-called "C rules for pointer operations" [5], we prefer to not introduce such a notion of "path memory safety" and, instead, to keep our notion of memory safety of programs in Definition 4; with it, these terminating programs are not memory safe. Later in this section we introduce a stronger notion of memory safety, supported by an executable semantics that will always get stuck on these programs.

Proposition 5 *Memory safety of terminating* KernelC *programs is an undecidable property.*

Proof. Since KernelC is Turing complete, we can encode any decidable property $\varphi(n)$ of input $n \in Nat$ as a *terminating* and *memory-safe* KernelC program "$\texttt{x=}n\texttt{;PGM}_\varphi$" which writes some variable `out`, such that $\varphi(n)$ holds iff KernelC $\models [\![\texttt{x=}n\texttt{;PGM}_\varphi]\!] \rightarrow^* \langle\langle\cdot\rangle_k\langle\texttt{out} \mapsto 1, ...\rangle_{env}...\rangle$ and $\varphi(n)$ does not hold iff KernelC $\models [\![\texttt{x=}n\texttt{;PGM}_\varphi]\!] \rightarrow^* \langle\langle\cdot\rangle_k\langle\texttt{out} \mapsto 0, ...\rangle_{env}...\rangle$. Since the pointer returned by `malloc` is non-deterministic, we can use it to "choose a random" n to assign to `x`: consider the program "$\texttt{PGM'}_\varphi \equiv \texttt{x=malloc(1);PGM}_\varphi\texttt{;if(out)GOOD else BAD}$". $\texttt{PGM'}_\varphi$ terminates because "$\texttt{x=}n\texttt{;PGM}_\varphi$" terminates for any $n \in Nat$ returned by `malloc(1)` and the conditional always terminates. On the other hand, $\texttt{PGM'}_\varphi$ is memory safe iff the variable `out` is 1 in the environment when $\texttt{PGM}_\varphi$ terminates, which happens iff $\varphi(n)$ holds for all $n \in Nat$. The undecidability of memory safety then follows from the fact that there are decidable properties φ for which $(\forall n)\varphi(n)$ is a proper co-recursively-enumerable property [17]. □

Since our notion of memory safety refers to a program rather than a path, the proposition above says that it is also impossible to devise any runtime checker for memory safety of general purpose KernelC (and hence C) programs. One could admittedly argue that such anomalies occur as artifacts of poorly designed languages like C, that allow for (too) direct memory access and complete freedom in handling pointers as if they are natural numbers. However, it is actually precisely these capabilities that make C attractive when performance is a concern, and performance is indeed a concern in many applications. That memory unsafe programs may execute just fine is a *must* feature of any formal semantic definition of C that is worth its salt, because all C implementations deliberately "suffer" from this problem.

Note that we are not attempting to fix C's problems here, nor to propose a better language design. Our goal is to propose a program verification approach based on matching and rewriting logic, and so, for this purpose, the fact that the base language is intricate is a plus. However, the high degree of non-determinism in the semantics of `malloc` may be problematic in formal verification. We prefer to give a slightly different semantics to our language, one which captures the non-determinism of `malloc` *symbolically*. Figure 4 shows the formal K semantic definition of SafeKernelC, which essentially adds symbolic numbers and gives `malloc` a symbolic semantics. Everything else stays unchanged, like in the definition of KernelC in Figure 2.

(abstract syntax)

NatVar ::= infinite set of symbolic natural numbers
Nat ::= ... | *NatVar*

(semantic equations and rules)

$\langle \texttt{malloc}(N)\texttt{;} \curvearrowright K\rangle_k \langle\sigma\rangle_{mem}\langle\pi\rangle_{ptr} \rightarrow \langle p \curvearrowright K\rangle_k\langle\sigma \circledast \sigma'\rangle_{mem}\langle\pi[P \leftarrow N]\rangle_{ptr}$
where P is a fresh symbol in *NatVar* and $Dom(\sigma') = \overline{P, P+N-1}$

Figure 4. Formal semantics of SafeKernelC. (figure only shows how it differs from the semantics of KernelC in Figure 2)

Definition 6 *Well-formed computation K is* ***strongly terminating*** *iff $[\![K]\!]$ is terminating in* SafeKernelC*, and is* ***strongly memory safe*** *iff any normal form of $[\![K]\!]$ in* SafeKernelC *is final.*

Since SafeKernelC adds symbolic values (for pointers and initial values in allocated memory locations), the assumed machinery for naturals and integers is now expected to work with these symbolic values as well. In particular, the rule (side) conditions may be harder to check. For example, the rule "$I_1\texttt{==}I_2 \rightarrow N$" applies only when one proves that $I_1 = I_2$, and in that case N is 1, or when one proves that $I_1 \neq I_2$, and in that case N is 0; if one cannot prove any of the two, then the term "$I_1\texttt{==}I_2$" remains unreduced and the execution of the program may get stuck because of that. For example, both "`p=malloc(1);while(*p){}`" and "`p=malloc(1);while(*1){}`" are now strongly terminating (but remain memory unsafe, also in the strong sense). Also, both programs discussed in front of Proposition 5 get stuck when processing the conditions of their `if` statements. On the positive side, programs obeying the recommended safety rules for pointer operations in C [5], e.g., reading only initialized locations and comparing pointers only if they are within the same data-structure contiguously allocated in memory, are strongly memory safe. For example, "`n=100;a=malloc(n);x=a;while(x!=a+n){*x=0;x=x+1;}`" is both strongly memory safe and strongly terminating.

Proposition 7 *Let $p \in K$ be a program. Then*

1. *If p is terminating then p is strongly terminating;*
2. *If p is strongly memory safe then p is memory safe;*

3. *If p is strongly memory safe then p is terminating iff p is strongly terminating.*

The first two implications in the proposition above are proper. For example, "`p=malloc(1);while(*p){}`" is a strongly terminating program (but not strongly memory safe) which is not terminating. We call such programs "accidentally non-terminating". There are also programs which are memory safe but not strongly memory safe, such as "`x=malloc(1);y=malloc(1);if(y==x+1){}else{}`". We call such programs "accidentally memory safe".

Proposition 8 *Strong termination and strong memory safety remain undecidable in general, but strong memory safety of strongly terminating programs is decidable.*

5. Matching Logic

Matching logic can be thought of both as a symbolic "big step" semantics associated to a K language definition and as a configuration-based axiomatic semantics of the language. Matching logic derives pairs $\Gamma \Downarrow \Gamma'$ of *configuration patterns*, called *(partial) correctness pairs*. We exemplify it by defining a formal system for KernelC.

Recall from Section 4 that we let (Σ, E) denote the algebraic specification of KernelC configurations, i.e., Σ contains all the configuration constructs (for bags, maps, etc.) and E contains all their defining equations (associativities, commutativities, etc.). Like in rewriting logic, everything we do from here on takes place *modulo equations in E*; we therefore take the freedom to write $t = t'$ instead of KernelC $\models t = t'$ for any terms t and t'. Also, recall from Definition 2 that the (concrete) configurations of KernelC are well-formed ground *Cfg*-terms of the form $\langle\langle K\rangle_k \langle\rho\rangle_{env} \langle\sigma\rangle_{mem} \langle\pi\rangle_{ptr}\rangle$. In other words, configurations have only concrete data (no variables) and we assume all their equational properties by default, in particular we can conveniently write their subcells in any order.

5.1. Patterns

Configuration patterns, or simply just *patterns*, play a central role in matching logic. Patterns take the place of the formulae in Hoare logics, so they are program state specifications; more precisely, patterns are program configuration specifications, i.e., they can also refer to programs themselves, not only to their state. Technically, patterns are configuration terms augmented with *variables* and *constraints* over them. Patterns can generalize by a mechanism called *pattern abstraction*, and relate to concrete program specifications by a mechanism called *pattern matching*; both these mechanisms are very natural and intuitive, yet they involve some technical details that need to be discussed.

Formulae

We here discuss the logical infrastructure needed later on to define the pattern constraints. Let *Form* be a new sort, for formulae, providing at least the following signature: *true* and *false* constants, a semantic entailment relation $\psi \models \varphi$ (i.e., operation of bool result; sort bool is assumed), and conjunction $\varphi \wedge \psi$. We also assume a mechanism for substitution of variables in formulae: $\overline{\theta}(\varphi)$ is the formula applying substitution θ to φ; of course, in case formulae have quantifiers or other kinds of binders, then substitutions are supposed to act on free variables only and in a capture-free manner. These operations are required to satisfy the following natural properties (φ, φ', ψ range over *Form*):

- *False hypothesis: false* $\models \varphi$;
- *True conclusion:* $\varphi \models$ *true*;
- *Reflectivity:* $\varphi \models \varphi$;
- *Transitivity:* If $\psi \models \varphi$ and $\varphi \models \varphi'$ then $\psi \models \varphi'$;
- *Substitution closure:* If $\psi \models \varphi$ then $\overline{\theta}(\psi) \models \overline{\theta}(\varphi)$ for any substitution θ.

We take the freedom to write $\models \varphi$ instead of *true* $\models \varphi$. Then, e.g., one can infer that if $\models \overline{\theta}(\varphi)$ and $\varphi \models \varphi'$, then $\models \overline{\theta}(\varphi')$. Substitutions play a major role in matching logic, because they can be the results of matching operations. We will define and discuss substitutions in detail shortly. For now, the reader can informally assume the usual meaning of a substitution, namely a mapping that associates terms to variables.

For mathematical uniformity, we assume that *Form* contains all the background signatures and theories of mathematical objects involved in proofs, including arithmetic, etc. Furthermore, we here also assume that *Form* includes equational logic, i.e., it has an equality formula construct for each sort or subsort in Σ, giving for any two terms t, t' of same sort S an equality formula $t =_S t'$. We drop the index S whenever clear from context. When implementing matching logic provers, for efficiency reasons, we do not advocate flattening all the logical infrastructure of a configuration into one formula; in our experience with our prover (Section 7), it can be more convenient to store axiomatizations of various mathematical domains of interest as background library theories, and to use a specialized equational/rewriting derivation engine with powerful support for matching "outside" of the logical infrastructure. However, we here chose to flatten everything in a formula because it simplifies the exposition, in that we can now uniformly assume that the formula embedded in a pattern contains all the logical ingredients needed to reason about that pattern.

It is worth mentioning here that matching logic will require overall less support from the underlying logic than Hoare logics. Indeed, the latter requires the logic to have existential quantifiers in order to state loop invariants. Matching logic will never require any quantifier support from the underlying logic; in particular, the loop invariants will make use of patterns' binding variables.

Patterns, Free and Bound Pattern Variables

We now define patterns as configuration terms with variables (slots for matching) embedding a subset of *bound* variables and a formula constraining all its variables.

Definition 9 *Consider a new sort Var and, for each existing sort S, a new subsort SVar < S together with infinitely many variable symbols of sort SVar; also, assume that SVar < Var. Consider two additional configuration item cells, namely:*

$$CfgItem ::= \ldots \mid \langle \mathsf{Bag}_{\_}\,\mathsf{'}\,{}_{\_}[Var]\rangle_{bnd} \mid \langle Form\rangle_{form}$$

For simplicity, we also use (Σ, E) for the extended algebraic specification of configurations and $\mathcal{T}$ for the (larger) Σ-algebra of ground terms. Let $\mathcal{T}(Var)$ be the Σ-algebra of terms with variables in Var. The E-equational classes of terms in $\mathcal{T}(Var)$ of sort Cfg which are 6-cell configuration of the form $\langle\langle K\rangle_k\,\langle\rho\rangle_{env}\,\langle\sigma\rangle_{mem}\,\langle\pi\rangle_{ptr}\,\langle V\rangle_{bnd}\,\langle\varphi\rangle_{form}\rangle$ are called ***(configuration) patterns****. If Γ is a pattern as above, then we call $bnd(\Gamma) = V$ the set of its* ***bound variables****, we call the set $vars(\Gamma)$ of all the variables that appear in Γ (except those bound in φ, if any) the* ***pattern variables*** *of Γ, we call $free(\Gamma) = vars(\Gamma) - bnd(\Gamma)$ the set of* ***free variables*** *of Γ, and finally call φ the* ***constraints*** *or the* ***formula*** *of Γ.*

In what follows, $\Gamma, \Gamma', \Gamma_1, \ldots$ range over patterns, $\gamma, \gamma', \gamma_1, \ldots$ range over concrete configurations, and $C, C', C_1, \ldots$ range over terms which are bags of configuration items. Unless otherwise stated directly or indirectly, the configuration item bag terms C, C', C_1, etc., can have variables.

We borrow the terminology of finality from configurations (see Definition 2) to patterns:

Definition 10 *Patterns $\langle\langle K\rangle_k \ldots\rangle$ whose embedded computation K is well-terminated (a "$\cdot$" or a term of sort Int) are called* ***final patterns****.*

The intuition for final patterns is that they specify concrete configurations which are themselves final, that is, they can not be rewritten anymore within the rewriting logic semantics of KernelC.

Substitutions

Substitutions typically act on a specific subset of variables, letting all the other variables unchanged. In matching logic, substitutions play a very crucial role: they are the results of successful matching operations. Since we want matching logic proofs to refer to finitely presentable mathematical objects, so that they can be produced and used by program verification tools, we found it more convenient to work with substitutions as finite domain maps, same like the other map structures considered in our language definitions:

Definition 11 *A* ***substitution*** *$\xi : V \to \mathcal{T}(Var)$ is a sort-preserving map ξ from a finite set of variables $V \subseteq Var$ to terms which can have variables. If ξ is such a substitution, then we may call it a V-****substitution*** *and/or may say that V is the set of variables on which ξ* ***acts****. When $\xi(v)$ is a ground term for each $v \in V$, we call ξ a* ***ground substitution****.*

Finally, we let $\overline{\xi} : \mathcal{T}(Var) \to \mathcal{T}(Var)$ denote the unique ***homomorphic extension*** *of $\xi : V \to \mathcal{T}(Var)$ to arbitrary terms which is the identity on all variables in $Var - V$.*

Note that if $\xi_1 : V_1 \to \mathcal{T}(Var)$ and $\xi_2 : V_2 \to \mathcal{T}(Var)$ are two substitutions, then both $\overline{\xi_1} \circ \xi_2 : V_2 \to \mathcal{T}(Var)$ and $\overline{\xi_2} \circ \xi_1 : V_1 \to \mathcal{T}(Var)$ are substitutions. However, note that it is not necessarily the case that the homomorphic extensions of substitutions have without additional constraints the "Kleisli" extension property $\overline{\overline{\xi_1} \circ \xi_2} = \overline{\xi_1} \circ \overline{\xi_2}$. For example, if $x_1, x_2 \in IntVar$ and ξ_1 is an $\{x_1\}$-substitution with $\xi_1(x_1) = 0$ and ξ_2 is an $\{x_2\}$-substitution with $\xi_2(x_2) = x_1$, then $\overline{\overline{\xi_1} \circ \xi_2}(x_1 + x_2) = x_1 + 0$ while $(\overline{\xi_1} \circ \overline{\xi_2})(x_1 + x_2) = 0 + 0$. The "additional constraints" needed to make the above can be to restrict the homomorphic extensions of substitutions to only terms over the variables on which the substitution acts. However, we will shortly see that in matching logic some substitutions act only on the free variables in a pattern but not on the bound ones, while other substitutions only act on the bound variables but not on the free ones.

For notational consistency, we will attempt to use $\tau, \tau', \tau_1, \ldots$ for ground substitutions, $\xi, \xi', \xi_1, \ldots$ for substitutions which are not necessarily ground but which typically only act on the free variables of the patterns on which they are applied, and $\theta, \theta', \theta_1, \ldots$ for substitutions of variables bound in the patterns on which they are applied. To avoid confusion, we are going to mention the type of substitutions whenever that is not clearly implied by the context.

Pattern Abstraction

To increase the modularity of program verification, we want to prove each task in a setting which is as general as possible and then instantiate that general setting to various special cases. That means, in particular, that we need a mechanism to generalize and/or refine specifications. Since in matching logic patterns specify program configurations, we need a mechanism to generalize and/or refine patterns. We here introduce *pattern abstraction*, written $\Gamma \Rrightarrow \Gamma'$, which will do precisely that; we say that Γ is "more concrete than" Γ', or that Γ' is "more abstract than" Γ.

Definition 12 *Pattern $\Gamma' = \langle C'\,\langle V'\rangle_{bnd}\,\langle\varphi'\rangle_{form}\rangle$* ***abstracts*** *$\Gamma = \langle C\,\langle V\rangle_{bnd}\,\langle\varphi\rangle_{form}\rangle$, or Γ* ***refines*** *Γ', written $\Gamma \Rrightarrow_{[\theta]} \Gamma'$ with subscript θ optional (i.e., mentioned only when needed in context), iff $free(\Gamma) \supseteq free(\Gamma')$ and $free(\Gamma) \cap bnd(\Gamma') = \emptyset$ and θ is a V'-substitution such that $C = \overline{\theta}(C')$ and $\varphi \models \overline{\theta}(\varphi')$.*

Pattern abstraction is therefore some kind of an abstract matching operation, in that if $\Gamma \Rrightarrow \Gamma'$ then Γ can be thought of as matching Γ' in a way that is consistent with their internal constraints. Note that the substitution θ used for pattern abstraction and appearing optionally as a subscript in the notation $\Gamma \Rrightarrow_\theta \Gamma'$ only acts on bound variables in Γ' and is not required to be ground; in fact, for a $v' \in V'$, the term $\theta(v')$ may contain both bound and free variables in Γ. When a pattern abstraction step $\Gamma \Rrightarrow_\theta \Gamma'$ is applied in a matching logic proof, the free variables of Γ are going to sooner or later be bound in a larger proof context; since θ only acts on the bound variables of Γ' and since $free(\Gamma) \supseteq free(\Gamma')$, the contextual bindings of the free variables in the abstracted pattern Γ' are going to remain the same as in the concrete pattern Γ, as expected. The condition $free(\Gamma) \cap bnd(\Gamma') = \emptyset$ has technical motivations, to avoid variable captures in the context of substitutions. As seen later in this section, the bound variables in patterns can be α-converted, so this condition is easy to ensure in matching logic proofs. The conditions $free(\Gamma) \supseteq free(\Gamma')$ and $free(\Gamma) \cap bnd(\Gamma') = \emptyset$ in the definition of pattern abstraction are stronger than needed in the subsequent proofs, but we prefer them because they are more compact to state and more intuitive than what is needed; besides, we have no practical need to weaken them yet. We assume, of course, that no variable capture occurs in $\overline{\theta}(\varphi')$; as already mentioned when we discussed the logic infrastructure, one should α-convert variables that φ' binds (if any) appropriately before applying $\overline{\theta}$ to φ'.

The most immediate operation that one can perform on a pattern is to "concretize" it via a substitution of its bound variables. However, one should make sure that the free/bound variable conditions in Definition 12 still hold:

Proposition 13 *If $\Gamma = \langle C \langle V \rangle_{bnd} \rangle$ and $\Gamma' = \langle C' \langle V' \rangle_{bnd} \rangle$ with $free(\Gamma) \supseteq free(\Gamma')$ and $free(\Gamma) \cap bnd(\Gamma') = \emptyset$, and if θ is a V'-substitution such that $C = \overline{\theta}(C')$, then $\Gamma \Rrightarrow_\theta \Gamma'$.*

Proof. The only difference between the hypothesis of this proposition and Definition 12 is that the pattern conditions are not mentioned; instead, if φ is the condition of Γ then the condition of Γ' is $\overline{\theta}(\varphi)$. Since $\overline{\theta}(\varphi) \models \overline{\theta}(\varphi)$, this proposition is therefore an immediate consequence of Definition 12. □

Particularly interesting instances of the proposition above are when θ is a bijection renaming V' into V (this case leads to α-conversion and is discussed below), or when θ is ground and $V = \emptyset$.

The finite set of bound variables can be arbitrarily extended in any pattern abstraction:

Proposition 14 *If $\Gamma \Rrightarrow_\theta \Gamma'$ and U is a finite set of variables disjoint from $vars(\Gamma) \cup vars(\Gamma')$, then $\Gamma_U \Rrightarrow_{\theta_U} \Gamma'_U$, where Γ_U and Γ'_U extend the set of bound variables in Γ and Γ' with U, respectively, and θ_U is the $(bnd(\Gamma') \cup U)$-substitution extending θ with the identity on the variables in U.*

Proof. Let $\Gamma = \langle C \langle V \rangle_{bnd} \langle \varphi \rangle_{form} \rangle$ and $\Gamma' = \langle C' \langle V' \rangle_{bnd} \langle \varphi' \rangle_{form} \rangle$ and θ a V' substitution such that $C' = \overline{\theta}(C)$ and $\varphi \models \overline{\theta}(\varphi')$. Then $\Gamma_U = \langle C \langle V, U \rangle_{bnd} \langle \varphi \rangle_{form} \rangle$ and $\Gamma'_U = \langle C' \langle V', U \rangle_{bnd} \langle \varphi' \rangle_{form} \rangle$, and $\overline{\theta_U}(C') = \overline{\theta}(C')$ and $\overline{\theta_U}(\varphi') = \overline{\theta}(\varphi')$. Therefore, $\Gamma_U \Rrightarrow_{\theta_U} \Gamma'_U$. □

Two patterns can abstract each other:

Definition 15 *Patterns Γ and Γ' are **equivalent** or **equally abstract**, written $\Gamma \Leftrightarrow \Gamma'$, iff $\Gamma \Rrightarrow \Gamma'$ and $\Gamma' \Rrightarrow \Gamma$.*

Pattern equivalence $\Gamma \Leftrightarrow \Gamma'$ implies, in particular, that $free(\Gamma) = free(\Gamma')$. It also subsumes two very practical operations, namely α-conversion (or renaming) and elimination/addition of redundant bound pattern variables:

Proposition 16 *If $\Gamma = \langle C \langle V \rangle_{bnd} \rangle$ and U is a finite set of variables disjoint from $vars(\Gamma)$, then:*

1. *$\Gamma \Leftrightarrow \langle \overline{\alpha}(C) \langle U \rangle_{bnd} \rangle$ if $\alpha : V \to U$ is a bijection;*
2. *$\Gamma \Leftrightarrow \langle C \langle V, U \rangle_{bnd} \rangle$.*

Proof. "*1.*" follows by noting that $\Gamma \Rrightarrow_{\alpha^{-1}} \langle \overline{\alpha}(C) \langle U \rangle_{bnd} \rangle$ and $\langle \overline{\alpha}(C) \langle U \rangle_{bnd} \rangle \Rrightarrow_\alpha \Gamma$. To prove "*2.*", for both abstractions involved take θ to be the identity substitution on V and anything on U; the idea here is that $\overline{\theta}(C) = C$ no matter how θ is defined on the variables in U. □

The pattern abstraction operation is transitive, that is, if $\Gamma_1 \Rrightarrow \Gamma_2$ and $\Gamma_2 \Rrightarrow \Gamma_3$ such that $bnd(\Gamma_3) \cap free(\Gamma_1) = \emptyset$ then $\Gamma_1 \Rrightarrow \Gamma_3$. More precisely,

Proposition 17 *If $\Gamma_1 \Rrightarrow_{\theta_1} \Gamma_2$ and $\Gamma_2 \Rrightarrow_{\theta_2} \Gamma_3$ such that $bnd(\Gamma_3) \cap free(\Gamma_1) = \emptyset$ then $\Gamma_1 \Rrightarrow_{\overline{\theta_1} \circ \theta_2} \Gamma_3$.*

Proof. Let us assume that Γ_1 is the pattern $\langle C_1 \langle V_1 \rangle_{bnd} \langle \varphi_1 \rangle_{form} \rangle$, Γ_2 is the pattern $\langle C_2 \langle V_2 \rangle_{bnd} \langle \varphi_2 \rangle_{form} \rangle$, and Γ_3 is the pattern $\langle C_3 \langle V_3 \rangle_{bnd} \langle \varphi_3 \rangle_{form} \rangle$, where $C_1 = \overline{\theta_1}(C_2)$ and $\varphi_1 \models \overline{\theta_1}(\varphi_2)$, where $C_2 = \overline{\theta_2}(C_3)$ and $\varphi_2 \models \overline{\theta_2}(\varphi_3)$, and where $free(\Gamma_1) \supseteq free(\Gamma_2) \supseteq free(\Gamma_3)$. Since variables in V_2 cannot be among the free variables of Γ_3, the above imply that all the occurrences of variables in V_2 that appear in $C_2 = \overline{\theta_2}(C_3)$ are actually produced by θ_2, that is, they appear as variables in a term $\theta_2(v_3)$ for some variable $v_3 \in V_3$ which occurs in C_3. That means that $(\overline{\overline{\theta_1} \circ \theta_2})(C_3) = \overline{\theta_1}(\overline{\theta_2}(C_3))$, that is, that $(\overline{\overline{\theta_1} \circ \theta_2})(C_3) = C_1$. Similarly, it follows that $(\overline{\overline{\theta_1} \circ \theta_2})(\varphi_3) = \overline{\theta_1}(\overline{\theta_2}(\varphi_3))$, and since the transitivity and substitution closure of $\models$ imply that $\varphi_1 \models \overline{\theta_1}(\overline{\theta_2}(\varphi_3))$, it follows that $\varphi_1 \models (\overline{\overline{\theta_1} \circ \theta_2})(\varphi_3)$. Therefore, $\Gamma_1 \Rrightarrow_{\overline{\theta_1} \circ \theta_2} \Gamma_3$. □

We next want to show that pattern abstraction is preserved by substitutions of free variables, that is, to show that if $\Gamma \Rrightarrow \Gamma'$ and ξ is some $free(\Gamma)$-substitution (recall that $free(\Gamma) \supseteq free(\Gamma')$), then $\overline{\xi}(\Gamma) \Rrightarrow \overline{\xi}(\Gamma')$. However, in order for this result to hold, some measures need to be taken to avoid "unexpected" variable captures. First, note that while the bound and free variables of Γ and Γ' are respectively disjoint, nothing is known about the disjointness of the free variables in Γ and the bound variables of Γ'; if they are not disjoint then $\overline{\xi}$ may wrongly act on bound variables in Γ' when applied to calculate $\overline{\xi}(\Gamma')$. Second, note that if the term $\xi(v)$ for some $v \in \mathit{free}(\Gamma)$ contains any variable which is bound by Γ or Γ', then that variable may be undesirably captured by the bound variables of $\overline{\xi}(\Gamma)$ and/or $\overline{\xi}(\Gamma')$.

Proposition 18 *Suppose that* $\Gamma \Rrightarrow_\theta \Gamma'$ *and that* $\xi : \mathit{free}(\Gamma) \to \mathcal{T}(\mathit{Var})$ *is a* $\mathit{free}(\Gamma)$*-substitution which does not interfere with the bound variables of* Γ *and* Γ'*, that is, the set* $\mathit{bnd}(\Gamma) \cup \mathit{bnd}(\Gamma')$ *is disjoint from the set of variables of* $\xi(v)$ *for any* $v \in \mathit{free}(\Gamma)$*. Then* $\overline{\xi}(\Gamma) \Rrightarrow_{\overline{\xi}\circ\theta} \overline{\xi}(\Gamma')$*.*

Proof. Suppose that Γ is the pattern $\langle C\,\langle V\rangle_{bnd}\,\langle\varphi\rangle_{form}\rangle$ and that Γ' is the pattern $\langle C'\,\langle V'\rangle_{bnd}\,\langle\varphi'\rangle_{form}\rangle$, so that θ is a V'-substitution with $C = \overline{\theta}(C')$ and $\varphi \models \overline{\theta}(\varphi')$. Then, since ξ does not interfere with the bound variables in Γ and Γ', it follows that $\overline{\xi}(\Gamma)$ is the pattern $\langle\overline{\xi}(C)\,\langle V\rangle_{bnd}\,\langle\overline{\xi}(\varphi)\rangle_{form}\rangle$ and $\overline{\xi}(\Gamma')$ is the pattern $\langle\overline{\xi}(C')\,\langle V'\rangle_{bnd}\,\langle\overline{\xi}(\varphi')\rangle_{form}\rangle$. Moreover, the non-interference hypothesis also implies that $(\overline{\overline{\xi}\circ\theta})(\overline{\xi}(C')) = \overline{\xi}(\overline{\theta}(C'))$ and that $(\overline{\overline{\xi}\circ\theta})(\overline{\xi}(\varphi')) = \overline{\xi}(\overline{\theta}(\varphi'))$. Therefore, $(\overline{\overline{\xi}\circ\theta})(\overline{\xi}(C')) = \overline{\xi}(C)$ and $\overline{\xi}(\varphi) \models (\overline{\overline{\xi}\circ\theta})(\overline{\xi}(\varphi'))$. Since it is also the case that $\mathit{free}(\overline{\xi}(\Gamma)) \supseteq \mathit{free}(\overline{\xi}(\Gamma'))$ and $\mathit{free}(\overline{\xi}(\Gamma)) \cap \mathit{bnd}(\overline{\xi}(\Gamma')) = \emptyset$, we conclude that indeed $\overline{\xi}(\Gamma) \Rrightarrow_{\overline{\xi}\circ\theta} \overline{\xi}(\Gamma')$. □

A simple case when the non-interference hypothesis of Proposition 18 holds trivially is when ξ is ground.

All the variable capture hypotheses in Proposition 18 can be easily achieved by applying, if necessary, α-conversions to Γ and/or Γ'. Thus, one can easily show the following:

Corollary 19 *If* $\Gamma \Rrightarrow \Gamma'$ *and* ξ *is an arbitrary* $\mathit{free}(\Gamma)$*-substitution, then there are configurations* Γ_0 *and* Γ'_0 *such that* $\Gamma \Leftrightarrow \Gamma_0$ *and* $\Gamma' \Leftrightarrow \Gamma'_0$ *and* $\overline{\xi}(\Gamma_0) \Rrightarrow \overline{\xi}(\Gamma'_0)$*.*

Proof. Let Γ_0 and Γ'_0 be some arbitrary α-converted variants of Γ and Γ', respectively. Since $\mathit{free}(\Gamma) = \mathit{free}(\Gamma_0)$, ξ satisfies the hypotheses of Proposition 18 for $\Gamma_0 \Rrightarrow \Gamma'_0$. □

Pattern Matching and Concrete Patterns

The relationship between patterns and configurations is established by what we call *pattern matching* in matching logic. Intuitively, configuration γ matches pattern Γ, written $\gamma \models \Gamma$, iff there is some ground substitution of the variables in Γ which is consistent with Γ's constraints and which transforms Γ into γ (forgetting the two additional cells $\langle ...\rangle_{bnd}$ and $\langle ...\rangle_{form}$ which, in this case, are redundant).

Definition 20 *Configuration* γ ***matches*** *pattern* Γ*, written* $\gamma \models_{[\tau]} \Gamma$ *with the index* τ *optional (written only when needed in context), iff* $\Gamma = \langle C\,\langle V\rangle_{bnd}\,\langle\varphi\rangle_{form}\rangle$ *and* $\gamma = \langle\overline{\tau}(C)\rangle$*, where* τ *is some ground substitution of* $\mathit{vars}(\Gamma)$ *such that* $\models \overline{\tau}(\varphi)$*.*

Pattern matching therefore makes no distinction between bound and free variables: all pattern variables are matched the same way. Both pattern abstraction and pattern matching are defined in terms of matching, the former via a substitution matching only the bound variables of the second pattern, while the latter via a ground substitution matching both the bound and the free variables in the pattern.

Definition 21 *Patterns of the form* $\langle C\,\langle\cdot\rangle_{bnd}\,\langle \mathit{true}\rangle_{form}\rangle$ *with* $\langle C\rangle$ *a (concrete) configuration, are called* ***concrete patterns****. If* $\gamma = \langle C\rangle$ *is a configuration, we let* $\widehat{\gamma}$ *denote the the concrete pattern* $\langle C\,\langle\cdot\rangle_{bnd}\,\langle \mathit{true}\rangle_{form}\rangle$ *associated to* γ*.*

As expected, pattern matching is a special case of pattern abstraction, namely an abstraction of the concrete pattern associated to the matched configuration to a pattern abstracting all the variables away:

Proposition 22 $\gamma \models_\tau \langle C\,\langle V\rangle_{bnd}\rangle$ *iff* $\widehat{\gamma} \Rrightarrow_\tau \langle C\,\langle \mathit{vars}(C)\rangle_{bnd}\rangle$*.*

Proof. Simple consequence of Definitions 20 and 12. □

Proposition 23 *If* $\gamma \models_\tau \Gamma$ *and* $\Gamma \Rrightarrow_\theta \Gamma'$ *then* $\gamma \models_{\tau'} \Gamma'$ *for some* τ' *such that* $\tau(x') = \tau'(x')$ *for any* $x' \in \mathit{free}(\Gamma')$*.*

Proof. It follows directly as a corollary of Propositions 22 and 14. However, because of the relevance of this result, we also give it a direct proof in what follows.

Let us suppose that $\Gamma' = \langle C'\,\langle V'\rangle_{bnd}\,\langle\varphi'\rangle_{form}\rangle$, that $\Gamma = \langle\overline{\theta}(C')\,\langle V\rangle_{bnd}\,\langle\varphi\rangle_{form}\rangle$ with $\varphi \models \overline{\theta}(\varphi')$, and that $\gamma = \langle\overline{\tau}(\overline{\theta}(C'))\rangle$ with $\models \overline{\tau}(\varphi)$. Let τ' then be the ground $\mathit{vars}(\Gamma')$-substitution defined as follows: $\tau'(x') = \tau(x')$ for all $x' \in \mathit{free}(\Gamma')$ and $\tau'(v') = \overline{\tau}(\theta(v'))$ for all $v' \in \mathit{bnd}(\Gamma') = V'$. It can be easily seen now that $\overline{\tau'}(C') = \overline{\tau}(\overline{\theta}(C'))$ and that $\overline{\tau'}(\varphi') = \overline{\tau}(\overline{\theta}(\varphi'))$, which imply the desired result. □

5.2. Correctness Pairs

In matching logic, configuration patterns are therefore program state specifications, a pattern specifying all those concrete configurations that match it. Also, as shown by Proposition 23, pattern abstraction *is* specification abstraction. However, unlike in Hoare logics but like in dynamic logic [6], the (fragment of) program itself is also part of the pattern. Hence, we introduce the following:

Definition 24 *A* ***(partial) correctness pair*** *is a pair of patterns written* $\Gamma \Downarrow \Gamma'$*, where* Γ' *is final.*

The intuition underlying $\Gamma \Downarrow \Gamma'$, captured formally by the *soundness* of matching logic (Theorem 27), is: if KernelC $\models \gamma \to^* \gamma'$ with γ' normal form and $\gamma \models \Gamma$, then also $\gamma' \models \Gamma'$, the two matchings agreeing on the free variables in Γ (and implicitly in Γ'; as seen in the sequel, the free variables of Γ will include those of Γ' whenever $\Gamma \Downarrow \Gamma'$ is derivable). The requirement that Γ' is final, that is that the computation K embedded in Γ' is either the unit computation "$\cdot$" or a term of sort *Int* says that K is a well-formed terminated computation, i.e., concrete configuration instances of Γ' cannot be rewritten anymore because they completed their intended task, not because they are stuck in some "unexpected" computation term.

We only discuss partial correctness here (i.e., we do not require that γ must terminate). When Γ' has an empty computation (i.e., the processed code was a statement), to simplify writing we use the following sugared notation, reminiscent to Hoare triples: $\langle C\rangle\, K\, \langle C'\rangle$ is $\langle\langle K\rangle_k\, C\rangle \Downarrow \langle\langle\cdot\rangle_k\, C'\rangle$.

5.3. Rules

Figure 5 shows the matching logic formal system associated to KernelC. One can systematically derive these rules from the K executable semantics in Figure 2, because they say the same thing but with different notations. For example, the rule for `malloc(`K`)` in Figure 5 says: first process K; if N is the result, then produce a "fresh" symbol P and update the configuration like in Figure 2, but adding P as a configuration parameter. Since `if` had two cases in the executable definition, we generate two proof obligations, one for each case. The rule for `while` is somehow similar to its Hoare logic variant (save for the possible side effect of its condition) and can also be (informally) derived from its executable definition in Figure 2.

The underlying ideas of the derivation of matching logic rules from executable rules are the following:

- Derive for each language construct the most general pre- and post-configurations on which the corresponding executable rule(s) match and apply;
- Move the side conditions of the executable rules as consequence of the embedded formula, e.g., if ψ is a side condition of an executable rule, then, in the corresponding pattern in its matching logic rule, "match" ... $\wedge\, \psi$ against its embedded formula;
- Add the variables appearing "fresh" in the right-hand-side terms of executable rules as pattern parameters;
- Generating two or more matching logic rule hypotheses for constructs whose executable semantics has multiple cases.

This mechanical process will be fully automated and proved sound elsewhere. We here only want to stress the fact that, since matching logic works with configurations inheriting the very same structure of the concrete configurations in the

$$\frac{\langle C\rangle\, K_1\, \langle C_1\rangle,\ \langle\langle K_2\rangle_k\, C_1\rangle \Downarrow \Gamma}{\langle\langle K_1\, K_2\rangle_k\, C\rangle \Downarrow \Gamma}$$

$$\frac{\langle\langle K_1\rangle_k\, C\rangle \Downarrow \langle\langle I_1\rangle_k\, C_1\rangle,\ \langle\langle K_2\rangle_k\, C_1\rangle \Downarrow \langle\langle I_2\rangle_k\, C_2\rangle}{\langle\langle K_1\ \texttt{op}\ K_2\rangle_k\, C\rangle \Downarrow \langle\langle I_1\ op_{Int}\ I_2\rangle_k\, C_2\rangle}$$

$$\frac{\begin{array}{c}\langle\langle K_1\rangle_k\, \langle\varphi\rangle_{form}\, C\rangle \Downarrow \langle\langle I_1\rangle_k\, \langle\varphi_1\rangle_{form}\, C_1\rangle,\\ \langle\langle K_2\rangle_k\, \langle\varphi_1 \wedge I_1 \neq 0\rangle_{form}\, C_1\rangle \Downarrow \Gamma,\\ \langle\langle K_3\rangle_k\, \langle\varphi_1 \wedge I_1 = 0\rangle_{form}\, C_1\rangle \Downarrow \Gamma\end{array}}{\langle\langle \texttt{if(}K_1\texttt{)}\ K_2\ \texttt{else}\ K_3\rangle_k\, \langle\varphi\rangle_{form}\, C\rangle \Downarrow \Gamma}$$

$$\langle\langle X\rangle_k\, \langle X \mapsto I, \rho\rangle_{env}\, C\rangle \Downarrow \langle\langle I\rangle_k\, \langle X \mapsto I, \rho\rangle_{env}\, C\rangle$$

$$\frac{\langle\langle K\rangle_k\, C\rangle \Downarrow \langle\langle I\rangle_k\, \langle\rho\rangle_{env}\, C'\rangle}{\langle C\rangle\, X{=}K;\, \langle\langle\rho[X \leftarrow I]\rangle_{env}\, C'\rangle}$$

$$\frac{\langle\langle K\rangle_k\, C\rangle \Downarrow \langle\langle P\rangle_k\, \langle P \mapsto I \circledast \sigma\rangle_{mem}\, C'\rangle}{\langle\langle {}^*K\rangle_k\, C\rangle \Downarrow \langle\langle I\rangle_k\, \langle P \mapsto I \circledast \sigma\rangle_{mem}\, C'\rangle}$$

$$\frac{\langle\langle K_1\rangle_k\, C\rangle \Downarrow \langle\langle P\rangle_k\, C_1\rangle,\ \langle\langle K_2\rangle_k\, C_1\rangle \Downarrow \langle\langle I\rangle_k\, \langle P \mapsto I' \circledast \sigma\rangle_{mem}\, C_2\rangle}{\langle C\rangle\, {}^*K_1{=}K_2;\, \langle\langle P \mapsto I \circledast \sigma\rangle_{mem}\, C_2\rangle}$$

$$\frac{\langle\langle K_1\rangle_k\, C\rangle \Downarrow \langle\langle I\rangle_k\, \langle\varphi\rangle_{form}\, C_1\rangle,\ \langle\langle\varphi \wedge I {\neq} 0\rangle_{form}\, C_1\rangle\, K_2\, \langle C\rangle}{\langle C\rangle\, \text{while}(K_1)\, K_2\, \langle\langle\varphi \wedge I{=}0\rangle_{form}\, C_1\rangle}$$

$$\frac{\langle\langle K\rangle_k\, C\rangle \ \Downarrow\ \langle\langle N\rangle_k\, \langle\sigma\rangle_{mem}\, \langle\pi\rangle_{ptr}\, \langle\varphi\rangle_{form}\, \langle V\rangle_{bnd}\, C'\rangle}{\langle\langle \texttt{malloc(}K\texttt{)}\rangle_k C\rangle \Downarrow \langle\langle P\rangle_k \langle\sigma \circledast \sigma'\rangle_{mem} \langle\pi[P \leftarrow N]\rangle_{ptr} \langle\varphi \wedge \psi\rangle_{form} \langle V, P\rangle_{bnd} C'\rangle}$$
$$\text{(where } \psi \text{ is } Dom(\sigma') = \overline{P, P+N-1}\text{)}$$

$$\frac{\langle\langle K\rangle_k\, C'\rangle \ \Downarrow\ \langle\langle P\rangle_k\, \langle\sigma \circledast \sigma'\rangle_{mem}\, \langle P \mapsto N, \pi\rangle_{ptr}\, \langle\varphi \wedge \psi\rangle_{form}\, C\rangle}{\langle\langle \texttt{free(}K\texttt{)}\rangle_k\, C'\rangle \ \Downarrow\ \langle\langle\cdot\rangle_k\, \langle\sigma\rangle_{mem}\, \langle\pi\rangle_{ptr}\, \langle\varphi\rangle_{form}\, C\rangle}$$
$$\text{(where } \psi \text{ is } Dom(\sigma') = \overline{P, P+N-1}\text{)}$$

Figure 5. KernelC**: Matching logic formal system**

executable K semantics, the two semantics are actually very close to each other. The automation of this will lead to generation of language proof systems *sound by construction*.

In addition to language specific rules, matching logic assumes the following general-purpose rules (to avoid adding side conditions, we assume by default that all pairs of configurations involved in general purpose matching logic rules are well-formed correctness pairs):

$$\textbf{\textit{Basic}}: \quad \begin{cases} \Gamma \Downarrow \Gamma & \text{when } \Gamma \text{ is final} \\ & \text{and} \\ \langle\langle \mathit{false}\rangle_{\mathit{form}}\, C\rangle \Downarrow \Gamma & \mathit{free}(C) \supseteq \mathit{free}(\Gamma) \end{cases}$$

$$\textbf{\textit{Consequence}}: \quad \frac{\Gamma_1 \Rrightarrow \Gamma_1',\ \Gamma_1' \Downarrow \Gamma_2',\ \Gamma_2' \Rrightarrow \Gamma_2}{\Gamma_1 \Downarrow \Gamma_2}$$

$$\textbf{\textit{Substitution}}: \quad \frac{\Gamma_1 \Downarrow \Gamma_2}{\bar{\xi}(\Gamma_1) \Downarrow \bar{\xi}(\Gamma_2)} \quad \begin{cases} \xi \text{ only acts on free} \\ \text{variables in } \Gamma_1 \text{ (and } \Gamma_2) \end{cases}$$

In the Substitution rule, ξ therefore only acts on variables that are free in Γ_1 (and Γ_2); also, we assume that ξ does not interfere with the bound variables of Γ_1 and Γ_2, that is, $\mathit{bnd}(\Gamma_1) \cup \mathit{bnd}(\Gamma_2)$ is disjoint from the set of variables of $\xi(v)$ for any $v \in \mathit{free}(\Gamma_1)$; in particular, ξ does not capture any of the bound variables in Γ_1 and/or Γ_2. Additionally, since pattern variables can be α-converted so there is no relationship between pattern variables happening to have the same name in Γ_1 and Γ_2 we also assume that $\mathit{free}(\Gamma_1) \cap \mathit{bnd}(\Gamma_2) = \emptyset$.

5.4. Framing

One can easily add framing rules for one's language if one wants to do so for certain configuration cells. In matching logic, almost every cell can yield a framing rule. However, not all of these rules are desirable or necessary in all languages, so we refrain from adding them by default. For example, a conventional memory framing rule would be unsound for a language defining a "safe exit" command which exits the program only when the memory is completely deallocated. Instead, we rather encourage the development of good proving methodologies allowing for framing by means of Substitution rule. Consider, e.g., the following correct matching logic proof of `SUM` (see page 1):

$\langle\langle \mathtt{p}\mapsto p\rangle_{env}\ \langle p \geq 0\rangle_{form}\ \langle\cdot\rangle_{bnd}\rangle$
`s=0;n=1;`$\langle\langle \mathtt{p}\mapsto p, \mathtt{s}\mapsto 0, \mathtt{n}\mapsto 1\rangle_{env}\ \langle p \geq 0\rangle_{form}\ \langle\cdot\rangle_{bnd}\rangle$
$\Rrightarrow [\theta(n) = 1]$
$\langle\langle \mathtt{p}\mapsto p, \mathtt{s}\mapsto n(n-1)/2, \mathtt{n}\mapsto n\rangle_{env}\ \langle p\geq 0 \wedge n \leq p+1\rangle_{form}\ \langle n\rangle_{bnd}\rangle$
`while(n!=p+1) {`
$\langle\langle \mathtt{p}\mapsto p, \mathtt{s}\mapsto n(n-1)/2, \mathtt{n}\mapsto n\rangle_{env}\ \langle p\geq 0 \wedge n < p+1\rangle_{form}\ \langle n\rangle_{bnd}\rangle$
`    s=s+n;  n=n+1;`
$\langle\langle \mathtt{p}\mapsto p, \mathtt{s}\mapsto n(n+1)/2, \mathtt{n}\mapsto n+1\rangle_{env}\ \langle p\geq 0 \wedge n < p+1\rangle_{form}\ \langle n\rangle_{bnd}\rangle$
$\Rrightarrow [\theta(n) = n + 1]$
$\langle\langle \mathtt{p}\mapsto p, \mathtt{s}\mapsto n(n-1)/2, \mathtt{n}\mapsto n\rangle_{env}\ \langle p\geq 0 \wedge n \leq p+1\rangle_{form}\ \langle n\rangle_{bnd}\rangle$
`}` $\langle\langle \mathtt{p}\mapsto p, \mathtt{s}\mapsto n(n-1)/2, \mathtt{n}\mapsto n\rangle_{env}\ \langle p\geq 0 \wedge n = p+1\rangle_{form}\ \langle n\rangle_{bnd}\rangle$
$\Rrightarrow [\theta(\rho) = (\mathtt{p}\mapsto p, \mathtt{n}\mapsto n), \theta(c) = (\langle p\geq 0 \wedge n = p+1\rangle_{form}\ \langle n\rangle_{bnd})]$
$\langle\langle \mathtt{s}\mapsto p(p+1)/2, \rho\rangle_{env}\ \langle \rho, c\rangle_{bnd}\ c\rangle$

Even though we proved that `s` eventually holds the sum of the first p numbers, the resulting correctness argument is, from a modularity perspective, rather poor: it can only be used in an environment containing only `p`, it cannot be composed with programs making use of the memory, it cannot even be sequentially composed with programs that need anything else from the environment but `s`, etc. A more general and modular way to prove this program is to derive the following (we let it as an exercise for the interested reader):

$\langle\langle \mathtt{p}\mapsto p, \rho\rangle_{env}\ \langle p \geq 0 \wedge \varphi\rangle_{form}\ \langle V\rangle_{bnd}\ c\rangle$
`s=0; n=1; while(n!=p+1) {s=s+n; n=n+1;}`
$\langle\langle \mathtt{p}\mapsto p, \mathtt{s}\mapsto p(p+1)/2, \mathtt{n}\mapsto p+1, \rho\backslash\mathtt{s}\backslash\mathtt{n}\rangle_{env}\ \langle p \geq 0 \wedge \varphi\rangle_{form}\ \langle V\rangle_{bnd}\ c\rangle$

Now one can prove the above correctness pair from this by applying a substitution ($\theta(\rho) = \cdot$, $\theta(\varphi) = \mathit{true}$, $\theta(V) = \cdot$, $\theta(c) = \cdot$) and then an abstraction. The free variables p, ρ, φ, V and c can be substituted to any corresponding terms, the last four thus possibly matching any corresponding frames.

5.5. Configuration Equations

In matching logic, equations (defined shortly) are regarded as structural identities, written using the symbol $\equiv$, and, like in rewriting logic [11], derivations in matching logic take place *modulo equations*:

$$\textbf{\textit{Modulo-}}\boldsymbol{\equiv}\textbf{\textit{:}} \quad \frac{\Gamma_1 \equiv \Gamma_1',\ \Gamma_1' \Downarrow \Gamma_2',\ \Gamma_2' \equiv \Gamma_2}{\Gamma_1 \Downarrow \Gamma_2}$$

As basic configuration equations, we inherit all the structural equations defining the configurations (cell structure equations, e.g., associativity, commutativity, etc.). Standard equational reasoning is also assumed. Equivalences of formulae embedded in configurations are regarded as equalities as well, and so are equational consequences of them. In other words, we assume the following:

$$\textbf{\textit{Basic-}}\boldsymbol{\equiv}\textbf{\textit{:}} \quad \frac{t = t'}{t \equiv t'}\ , \quad \text{plus full equational reasoning for } \equiv$$

Formula-≡: $\dfrac{\varphi \models \psi,\ \psi \models \varphi}{\langle\varphi\rangle_{form} \equiv \langle\psi\rangle_{form}}$

Context-≡: $Cxt[t]\ \langle t = t' \wedge \varphi\rangle_{form} \equiv Cxt[t']\ \langle t = t' \wedge \varphi\rangle_{form}$

for any context Cxt

One can add one's own equations to a matching logic formal system; however, to preserve soundness, the custom equations must be consistent (we discuss consistency shortly). Immediate candidates are equations making configuration properties explicit, e.g., maps induce disjointness:

$$\langle p \mapsto u \circledast q \mapsto v \circledast \sigma\rangle_{mem} \langle\varphi\rangle_{form} \equiv \langle p \mapsto u \circledast q \mapsto v \circledast \sigma\rangle_{mem} \langle p \neq q \wedge \varphi\rangle_{form}$$

In our prover based on rewriting logic, we do *not* add equations like the one above. We prefer to keep the formulae small. Instead, we derive $p \neq q$ when needed by a matching operation on the map. However, other provers may prefer to have all configuration constraints in one place.

However, what makes pattern equational reasoning interesting is that one is allowed to extend the signature of configurations and add one's own pattern equational definitions. For example, one can define a heap construct *list* taking a pointer and a sequence of integers, together with the following two pattern equations:

$$\langle list(p,\alpha) \circledast \sigma\rangle_{mem}\ \langle p{=}0 \wedge \varphi\rangle_{form} \equiv \langle\sigma\rangle_{mem}\ \langle p{=}0 \wedge \alpha{=}\epsilon \wedge \varphi\rangle_{form}$$

$$\begin{aligned}\langle list(p,\alpha) \circledast \sigma\rangle_{mem}\ \langle p{\neq}0 \wedge \varphi\rangle_{form}\ \langle V\rangle_{bnd}\ \langle\pi\rangle_{ptr} \equiv&\\ \langle V, a, q, \beta, \pi'\rangle_{bnd}\ \langle p \mapsto a \circledast p{+}1 \mapsto q \circledast list(q,\beta) \circledast \sigma\rangle_{mem}&\\ \langle p \neq 0 \wedge \alpha = a.\beta \wedge \pi = p \mapsto 2, \pi' \wedge \varphi\rangle_{form}\ \langle\pi\rangle_{ptr}&\end{aligned}$$

In the second equation, a, q, β are variable names that do not appear in the left hand side of the equation. The two equations above, together with pattern abstraction, allow us to identify sequences of integers as "mathematical objects" flattened in patterns; e.g., if $\pi = (3 \mapsto 2, 5 \mapsto 2)$ then:

$$\begin{aligned}&\langle 3 \mapsto 1 \circledast 4 \mapsto 0 \circledast 5 \mapsto 2 \circledast 6 \mapsto 3\rangle_{mem}\ \langle true\rangle_{form}\ \langle\cdot\rangle_{bnd}\ \langle\pi\rangle_{ptr} \equiv\\ &\langle list(0,\epsilon) \circledast 3 \mapsto 1 \circledast 4 \mapsto 0 \circledast 5 \mapsto 2 \circledast 6 \mapsto 3\rangle_{mem}\ \langle true\rangle_{form}\ \langle\cdot\rangle_{bnd}\ \langle\pi\rangle_{ptr} \Rrightarrow\\ &\langle list(q,\beta) \circledast 3 \mapsto a \circledast 4 \mapsto q \circledast 5 \mapsto 2 \circledast 6 \mapsto 3\rangle_{mem} \langle 1{=}a.\beta\rangle_{form} \langle a,q,\beta\rangle_{bnd} \langle\pi\rangle_{ptr}\\ &\equiv \langle list(3,1) \circledast 5 \mapsto 2 \circledast 6 \mapsto 3\rangle_{mem}\ \langle true\rangle_{form}\ \langle\cdot\rangle_{bnd}\ \langle\pi\rangle_{ptr} \Rrightarrow\\ &\langle list(q,\beta) \circledast 5 \mapsto a \circledast 6 \mapsto q\rangle_{mem}\ \langle 2.1 = a.\beta\rangle_{form}\ \langle a,q,\beta\rangle_{bnd}\ \langle\pi\rangle_{ptr} \equiv\\ &\langle list(5, 2.1)\rangle_{mem}\ \langle true\rangle_{form}\ \langle\cdot\rangle_{bnd}\ \langle\pi\rangle_{ptr}\end{aligned}$$

It is now a simple exercise, tedious but mechanical, to derive correctness proofs like the one in Figure 6 (to remove notational clutter, we did not include the frame variables and the $\langle ...\rangle_{ptr}$ cell) for list reverse, which would be very hard to derive in Hoare logic even for simpler languages than KernelC [15]. The hardest part is to guess the loop invariant, the rest of the steps are mechanical; our prover fills them automatically. For example, here is the detailed proof of the last abstraction step in the `while` body in Figure 6:

$$\left\langle \begin{array}{l}\langle \mathtt{p} \mapsto r, \mathtt{x} \mapsto r', \mathtt{y} \mapsto r', \rho'\rangle_{env} \langle list(q,\beta) \circledast r \mapsto c \circledast r{+}1 \mapsto q \circledast list(r',\gamma')\rangle_{mem}\\ \langle r{\neq}0 \wedge rev(\alpha) = rev(\gamma)\beta \wedge \gamma = c\gamma'\rangle_{form}\ \langle q, r, \rho', \beta, \gamma, c, \gamma'\rangle_{bnd}\end{array}\right\rangle$$

$\Rrightarrow$ [α-conversion δ with $\delta(c') = c,\ \delta(\beta') = \beta$]

$$\left\langle \begin{array}{l}\langle \mathtt{p} \mapsto r, \mathtt{x} \mapsto r', \mathtt{y} \mapsto r', \rho'\rangle_{env} \langle list(q,\beta') \circledast r \mapsto c' \circledast r{+}1 \mapsto q \circledast list(r',\gamma')\rangle_{mem}\\ \langle r{\neq}0 \wedge rev(\alpha){=}rev(\gamma)\beta \wedge \gamma{=}c\gamma' \wedge c\beta = c'\beta'\rangle_{form} \langle q, r, \rho', \beta, \gamma, c\gamma', c', \beta'\rangle_{bnd}\end{array}\right\rangle$$

$$\equiv \left\langle \begin{array}{l}\langle \mathtt{p} \mapsto r, \mathtt{x} \mapsto r', \mathtt{y} \mapsto r', \rho'\rangle_{env} \langle list(r, c\beta) \circledast list(r',\gamma')\rangle_{mem}\\ \langle r{\neq}0 \wedge rev(\alpha){=}rev(\gamma)\beta \wedge \gamma{=}c\gamma'\rangle_{form} \langle r, \rho', \beta, \gamma, c\gamma'\rangle_{bnd}\end{array}\right\rangle$$

$\Rrightarrow$ [$\theta(q) = r,\ \theta(r) = r',\ \theta(\rho) = (\mathtt{y} \mapsto r', \rho'),\ \theta(\beta) = c\beta,\ \theta(\gamma) = \gamma'$]

$$\left\langle \begin{array}{l}\langle \mathtt{p} \mapsto q, \mathtt{x} \mapsto r, \rho\rangle_{env} \langle list(q,\beta) \circledast list(r,\gamma)\rangle_{mem}\\ \langle rev(\alpha){=}rev(\gamma)\beta\rangle_{form} \langle q, r, \rho, \beta, \gamma\rangle_{bnd}\end{array}\right\rangle$$

In the last abstraction, $\theta(rev(\alpha){=}rev(\gamma)\beta)$ is indeed implied by $rev(\alpha){=}rev(\gamma)\beta \wedge \gamma{=}c\gamma'$ because $rev(c\gamma') = rev(\gamma')c$.

One can also derive the following (programs on page 1):

$$\langle\langle \mathtt{ALLOCATE}\rangle_k\ \langle\cdot\rangle_{env}\ \langle\cdot\rangle_{mem}\ \langle\cdot\rangle_{ptr}\ \langle\cdot\rangle_{bnd}\ \langle true\rangle_{form}\rangle$$
$$\Downarrow \left\langle \begin{array}{l}\langle\cdot\rangle_k \langle \mathtt{p} \mapsto p, \rho\rangle_{env} \langle list(p, 4.3.2.1.0)\rangle_{mem}\\ \langle\pi\rangle_{ptr} \langle p, \rho, \pi\rangle_{bnd} \langle true\rangle_{form}\end{array}\right\rangle$$

$$\left\langle \begin{array}{l}\langle \mathtt{DEALLOCATE}\rangle_k\ \langle \mathtt{p} \mapsto p, \mathtt{q} \mapsto q\rangle_{env}\ \langle list(p,\alpha) \circledast \sigma\rangle_{mem}\\ \langle\pi\rangle_{ptr}\ \langle\cdot\rangle_{bnd}\ \langle true\rangle_{form}\end{array}\right\rangle$$
$$\Downarrow \langle\langle\cdot\rangle_k\ \langle \mathtt{p} \mapsto 0, \mathtt{q} \mapsto 0\rangle_{env}\ \langle\sigma\rangle_{mem}\ \langle\pi\rangle_{ptr}\ \langle\pi\rangle_{bnd}\ \langle true\rangle_{form}\rangle$$

In experiments with our matching logic prover (see Appendix 7), we defined several other constructs besides *list*, including trees, queues, stacks, graphs, as well as parametric variants of them, e.g., stacks of trees, etc., and used them to verify several non-trivial programs, including the Schorr-Waite algorithm; details about all these will appear elsewhere.

Before we discuss consistency of pattern equations and soundness of matching logic in detail, we introduce a preliminary notion of well-formedness of pattern equations:

Definition 25 *A pattern equational specification is* ***well-formed*** *whenever the following hold:*

1. *If* $\Gamma \equiv \Gamma'$ *then* $free(\Gamma) = free(\Gamma')$*;*
2. *Each ≡-equation refers only to non-computational items of the pattern, that is, they do not contain any computation cell subterms.*

Well-formedness of pattern equational specification is a simple to check syntactic criterion: each equation must have the same free variables in both its terms, and non or the two terms should contain any $\langle ...\rangle_k$ cell as a subterm. This well-formedness criterion may not be the most relaxed one, but, however, all the pattern equations we were interested in so far verified these conditions, so we have no practical motivation for a more relaxed notion.

$\langle\langle \mathrm{p}\mapsto p\rangle_{env}\ \langle list(p,\alpha)\rangle_{mem}\ \langle true\rangle_{form}\ \langle p\rangle_{bnd}\rangle$

`if (p != null) {`$\langle\langle \mathrm{p}\mapsto p\rangle_{env}\ \langle list(p,\alpha)\rangle_{mem}\ \langle p\neq 0\rangle_{form}\ \langle p\rangle_{bnd}\rangle$

$\langle\langle \mathrm{p}\mapsto p\rangle_{env}\langle p\mapsto a\circledast p{+}1\mapsto p'\circledast list(p',\alpha')\rangle_{mem}\langle p\neq 0\wedge\alpha=a\alpha'\rangle_{form}\langle p,a,p',\alpha'\rangle_{bnd}\rangle$

`  x=*(p+1);`

$\langle\langle \mathrm{p}\mapsto p,\mathrm{x}\mapsto p'\rangle_{env}\langle p\mapsto a\circledast p{+}1\mapsto p'\circledast list(p',\alpha')\rangle_{mem}\langle p\neq 0\wedge\alpha=a\alpha'\rangle_{form}\langle p,a,p',\alpha'\rangle_{bnd}\rangle$

`  *(p+1)=null;`

$\langle\langle \mathrm{p}\mapsto p,\mathrm{x}\mapsto p'\rangle_{env}\langle p\mapsto a\circledast p{+}1\mapsto 0\circledast list(p',\alpha')\rangle_{mem}\langle p\neq 0\wedge\alpha=a\alpha'\rangle_{form}\langle p,a,p',\alpha'\rangle_{bnd}\rangle$

$\Rrightarrow [\theta(q)=p,\ \theta(r)=p',\ \theta(\rho)=\cdot,\ \theta(\beta)=a,\ \theta(\gamma)=\alpha']$

$\langle\langle \mathrm{p}\mapsto q,\mathrm{x}\mapsto r,\rho\rangle_{env}\langle list(q,\beta)\circledast list(r,\gamma)\rangle_{mem}\langle rev(\alpha)=rev(\gamma)\beta\rangle_{form}\langle q,r,\rho,\beta,\gamma\rangle_{bnd}\rangle$

`  while (x != null) {`

$\langle\langle \mathrm{p}\mapsto q,\mathrm{x}\mapsto r,\rho\rangle_{env}\langle list(q,\beta)\circledast list(r,\gamma)\rangle_{mem}\langle r\neq 0\wedge rev(\alpha)=rev(\gamma)\beta\rangle_{form}\langle q,r,\rho,\beta,\gamma\rangle_{bnd}\rangle$

$\left\langle\begin{matrix}\langle \mathrm{p}\mapsto q,\mathrm{x}\mapsto r,\rho\rangle_{env}\langle list(q,\beta)\circledast r\mapsto c\circledast r{+}1\mapsto r'\circledast list(r',\gamma')\rangle_{mem}\\ \langle r\neq 0\wedge rev(\alpha)=rev(\gamma)\beta\wedge\gamma=c\gamma'\rangle_{form}\ \langle q,r,\rho,\beta,\gamma,c,\gamma'\rangle_{bnd}\end{matrix}\right\rangle$

`    y=*(x+1); *(x+1)=p;`

$\left\langle\begin{matrix}\langle \mathrm{p}\mapsto q,\mathrm{x}\mapsto r,\mathrm{y}\mapsto r',\rho'\rangle_{env}\langle list(q,\beta)\circledast r\mapsto c\circledast r{+}1\mapsto q\circledast list(r',\gamma')\rangle_{mem}\\ \langle r\neq 0\wedge rev(\alpha)=rev(\gamma)\beta\wedge\gamma=c\gamma'\rangle_{form}\ \langle q,r,\rho',\beta,\gamma,c,\gamma'\rangle_{bnd}\end{matrix}\right\rangle$

`    p=x; x=y;`

$\left\langle\begin{matrix}\langle \mathrm{p}\mapsto r,\mathrm{x}\mapsto r',\mathrm{y}\mapsto r',\rho'\rangle_{env}\langle list(q,\beta)\circledast r\mapsto c\circledast r{+}1\mapsto q\circledast list(r',\gamma')\rangle_{mem}\\ \langle r\neq 0\wedge rev(\alpha)=rev(\gamma)\beta\wedge\gamma=c\gamma'\rangle_{form}\ \langle q,r,\rho',\beta,\gamma,c,\gamma'\rangle_{bnd}\end{matrix}\right\rangle$

$\Rrightarrow [\theta(q)=r,\ \theta(r)=r',\ \theta(\rho)=(\mathrm{y}\mapsto r',\rho'),\ \theta(\beta)=c\beta,\ \theta(\gamma)=\gamma']$

$\langle\langle \mathrm{p}\mapsto q,\mathrm{x}\mapsto r,\rho\rangle_{env}\langle list(q,\beta)\circledast list(r,\gamma)\rangle_{mem}\langle rev(\alpha)=rev(\gamma)\beta\rangle_{form}\langle q,r,\rho,\beta,\gamma\rangle_{bnd}\rangle$

`  }`

$\langle\langle \mathrm{p}\mapsto q,\mathrm{x}\mapsto r,\rho\rangle_{env}\langle list(q,\beta)\circledast list(r,\gamma)\rangle_{mem}\langle r=0\wedge rev(\alpha)=rev(\gamma)\beta\rangle_{form}\langle q,r,\rho,\beta,\gamma\rangle_{bnd}\rangle$

$\langle\langle \mathrm{p}\mapsto q,\mathrm{x}\mapsto r,\rho\rangle_{env}\langle list(q,\beta)\rangle_{mem}\langle r=0\wedge\gamma=\epsilon\wedge rev(\alpha)=rev(\gamma)\beta\rangle_{form}\ \langle q,r,\rho,\beta,\gamma\rangle_{bnd}\rangle$

$\Rrightarrow [\theta(p)=q,\ \theta(\rho)=(\mathrm{x}\mapsto r,\rho)]$

$\langle\langle \mathrm{p}\mapsto p,\rho\rangle_{env}\langle list(p,rev(\alpha))\rangle_{mem}\ \langle true\rangle_{form}\ \langle p,\rho\rangle_{bnd}\rangle$

`} else {`$\langle\langle \mathrm{p}\mapsto p\rangle_{env}\ \langle list(p,\alpha)\rangle_{mem}\ \langle p=0\rangle_{form}\ \langle p\rangle_{bnd}\rangle$

$\langle\langle \mathrm{p}\mapsto p\rangle_{env}\ \langle\cdot\rangle_{mem}\ \langle p=0\wedge\alpha=\epsilon\rangle_{form}\ \langle p\rangle_{bnd}\rangle$

$\Rrightarrow \langle\langle \mathrm{p}\mapsto p,\rho\rangle_{env}\langle list(p,rev(\alpha))\rangle_{mem}\ \langle true\rangle_{form}\ \langle p,\rho\rangle_{bnd}\rangle$

`}` $\qquad\qquad\langle\langle \mathrm{p}\mapsto p,\rho\rangle_{env}\langle list(p,rev(\alpha))\rangle_{mem}\ \langle true\rangle_{form}\ \langle p,\rho\rangle_{bnd}\rangle$

Figure 6. Matching logic proof of list reverse

5.6. Consistency and Soundness

Therefore, there are three types of pattern equations: (1) ones implicit in the definition of configurations (bags, maps, etc.); (2) ones defining configuration constructs useful for proving (lists, trees, graphs, etc.); and (3) ones defining mathematical objects, such as actual sequences with equationally defined operators such as *rev* with $rev(\epsilon) = \epsilon$ and $rev(a\alpha) = rev(\alpha)a$, trees, graphs, etc. All these may yield inconsistent pattern equational specifications.

Definition 26 *A pattern specification is* ***consistent*** *iff it is well-formed and the following hold (γ, γ' configurations):*

1. *It is not the case that $\widehat{\gamma}\ (\equiv\cup\Rrightarrow)^*\ \langle\langle false\rangle_{form}\ \_\rangle$.*
2. *If $\widehat{\gamma}\ (\equiv\cup\Rrightarrow)^*\ \widehat{\gamma'}$ then $\gamma = \gamma'$;*

The conditions above say that one cannot use the $\equiv$ equations and abstraction to generate an infeasible specification or to collapse otherwise distinct configuration concrete data. Proving consistency is an interesting but non-trivial subject, which we do not investigate here. It is the most general semantic hypothesis that we were able to find in order to prove the soundness of matching logic:

Theorem 27 ***(Soundness of matching logic)*** *Assume a consistent pattern specification for* KernelC *and suppose that $\widehat{\gamma}\Downarrow\widehat{\gamma'}$ is derivable, where γ and γ' are two (concrete) configurations. Then γ is memory safe and, if γ terminates then γ' is the only normal form configuration such that* KernelC $\models \gamma \rightarrow^* \gamma'$.

Proof. Since one typically derives correctness pairs whose patterns are not concrete, one may need to apply a Substitution step to make the desired correctness pair fit the hypothesis of Theorem 27. We stated this theorem in terms of concrete patterns instead of arbitrary patterns just for simplicity, to avoid formalizing the elimination of the additional configuration constructs, which are useful for proofs but are not part of concrete configurations. To prove this soundness result, however, we cannot avoid that elimination step and thus prove a more general result (lemma below). Before we do that, note that it is a simple inductive exercise to show that if $\Gamma\Downarrow\Gamma'$ is derivable then $free(\Gamma) \supseteq free(\Gamma')$.

Lemma 28 *Under the same consistency hypothesis of Theorem 27, suppose that correctness pair $\Gamma\Downarrow\Gamma'$ is derivable, with Γ and Γ' not necessarily concrete patterns, and that τ is a ground substitution of all free variables in Γ (and Γ').*

If $\widehat{\gamma}\ (\equiv\cup\Rrightarrow)^*\ \overline{\tau}(\Gamma)$ for some concrete configuration γ then γ is memory safe and, if γ terminates then there is a unique normal form configuration γ' with KernelC $\models \gamma \rightarrow^* \gamma'$*, and this unique γ' has the property that $\widehat{\gamma'}\ (\equiv\cup\Rrightarrow)^*\ \overline{\tau}(\Gamma')$.*

Let us first note that this result is indeed more general than the one stated in the theorem. Indeed, suppose that $\widehat{\gamma}\Downarrow\widehat{\gamma'}$ is derivable, where γ and γ' are two configurations and take $\Gamma = \widehat{\gamma}$, $\Gamma' = \widehat{\gamma'}$ and τ the empty substitution in Lemma 28. Since $\overline{\tau}(\Gamma) = \Gamma = \widehat{\gamma}$ and $\overline{\tau}(\Gamma') = \Gamma' = \widehat{\gamma'}$, Lemma 28 implies that γ is memory safe and, if it terminates then it has a unique normal form γ'', and that unique γ'' has the property that that $\widehat{\gamma''}\ (\equiv\cup\Rrightarrow)^*\ \widehat{\gamma'}$. Then the pattern specification consistency hypothesis (Definition 26) implies that $\gamma' = \gamma''$, so Theorem 27 holds.

Let us now prove Lemma 28. The proof proceeds by structural induction on the derivation tree of $\Gamma\Downarrow\Gamma'$. We first prove the soundness of the general purpose rules:

Basic: For the basic matching logic derivation rule $\Gamma\Downarrow\Gamma$ with Γ a final pattern, let τ be a ground substitution and let γ be a concrete configuration such that $\widehat{\gamma}(\equiv\cup\Rrightarrow)^*\ \overline{\tau}(\Gamma)$. Since Γ is final, that is its computation is well-terminated, the computation of $\overline{\tau}(\Gamma)$ is also well-terminated; moreover, since consistency implies that the computation structure is not altered by user-defined $\equiv$-equations, it follows that the computation of γ is also well-terminated, which means by Definition 1 that γ is a final configuration of KernelC; in particular, it is both memory safe and terminating, and it is its own unique normal form (i.e., $\gamma' = \gamma$ in Lemma 28). Then the result follows in a straightforward way.

The soundness of the basic derivation rule $\langle\langle \mathit{false}\rangle_{\mathit{form}}\, C\rangle \Downarrow \Gamma$ follows by default from the consistency hypothesis, because there is no concrete configuration γ such that $\widehat{\gamma}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\langle\langle \mathit{false}\rangle_{\mathit{form}}\, C\rangle) = \langle\langle \mathit{false}\rangle_{\mathit{form}}\,\overline{\tau}(C)\rangle$.

Consequence: $\dfrac{\Gamma_1 \Rrightarrow \Gamma_1',\ \Gamma_1' \Downarrow \Gamma_2',\ \Gamma_2' \Rrightarrow \Gamma_2}{\Gamma_1 \Downarrow \Gamma_2}$

Suppose that $\Gamma_1' \Downarrow \Gamma_2'$ satisfies the property in Lemma 28 and let us show that so does $\Gamma_1 \Downarrow \Gamma_2$. Let τ be a ground $\mathit{free}(\Gamma_1)$-substitution and γ a configuration such that $\widehat{\gamma}\,(\equiv \cup \Rrightarrow)^*\,\overline{\tau}(\Gamma_1)$. Let τ' be the ground $\mathit{free}(\Gamma_1')$-substitution with $\tau'(x') = \tau(x')$ for each $x' \in \mathit{free}(\Gamma_1')$, and note that $\overline{\tau}(\Gamma_1') = \overline{\tau'}(\Gamma_1')$ and $\overline{\tau'}(\Gamma_2) = \overline{\tau}(\Gamma_2)$. By Proposition 18 it follows that $\overline{\tau}(\Gamma_1) \Rrightarrow \overline{\tau}(\Gamma_1')$. Therefore, $\widehat{\gamma}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau'}(\Gamma_1')$. By the induction hypothesis we first conclude that γ is memory safe. Also, if γ terminates then there is a unique normal form γ' with KernelC $\models \gamma \to^* \gamma'$; moreover, this unique γ' has the property that $\widehat{\gamma'}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau'}(\Gamma_2')$. By Proposition 18 again it follows that $\overline{\tau'}(\Gamma_2') \Rrightarrow \overline{\tau'}(\Gamma_2)$. Since $\overline{\tau'}(\Gamma_2) = \overline{\tau}(\Gamma_2)$, we conclude that $\widehat{\gamma'}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\Gamma_2)$.

Substitution: $\dfrac{\Gamma_1 \Downarrow \Gamma_2}{\overline{\xi}(\Gamma_1) \Downarrow \overline{\xi}(\Gamma_2)}$ $\begin{cases} \xi \text{ only acts on free} \\ \text{variables in } \Gamma_1 \text{ (and } \Gamma_2) \end{cases}$

Suppose that $\Gamma_1 \Downarrow \Gamma_2$ satisfies the property in Lemma 28 and let us show that so does $\overline{\xi}(\Gamma_1) \Downarrow \overline{\xi}(\Gamma_2)$, where ξ is a $\mathit{free}(\Gamma_1)$-substitution satisfying all the requirements of the Substitution rule. Let τ be a ground $\mathit{free}(\overline{\xi}(\Gamma_1))$-substitution and γ a configuration such that $\widehat{\gamma}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\overline{\xi}(\Gamma_1))$. Let us consider the $\mathit{free}(\Gamma_1)$-substitution $\overline{\tau}\circ\xi$. One immediate observation is that $\overline{\tau}(\overline{\xi}(\Gamma_1)) = \overline{(\overline{\tau}\circ\xi)}(\Gamma_1)$ and $\overline{\tau}(\overline{\xi}(\Gamma_2)) = \overline{(\overline{\tau}\circ\xi)}(\Gamma_2)$. Therefore, $\widehat{\gamma}\,(\equiv\cup\Rrightarrow)^*\,\overline{(\overline{\tau}\circ\xi)}(\Gamma_1)$. By the induction hypothesis we obtain fist that γ is memory safe, and second that if γ terminates then it has a unique normal form, say γ', and $\widehat{\gamma'}\,(\equiv\cup\Rrightarrow)^*\,\overline{(\overline{\tau}\circ\xi)}(\Gamma_2)$. Since $\overline{\tau}(\overline{\xi}(\Gamma_2)) = \overline{(\overline{\tau}\circ\xi)}(\Gamma_2)$, we therefore conclude that $\widehat{\gamma'}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\overline{\xi}(\Gamma_2))$.

Modulo-≡: $\dfrac{\Gamma_1 \equiv \Gamma_1',\ \Gamma_1' \Downarrow \Gamma_2',\ \Gamma_2' \equiv \Gamma_2}{\Gamma_1 \Downarrow \Gamma_2}$

Suppose that $\Gamma_1' \Downarrow \Gamma_2'$ satisfies the property in Lemma 28 and let us show that so does $\Gamma_1 \Downarrow \Gamma_2$. Let τ be a ground $\mathit{free}(\Gamma_1)$-substitution and γ a configuration such that $\widehat{\gamma}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\Gamma_1)$. Since $\mathit{free}(\Gamma_1) = \mathit{free}(\Gamma_2)$ and $\equiv$ is closed under equational reasoning, it follows that $\overline{\tau}(\Gamma_1) \equiv \overline{\tau}(\Gamma_1')$. Therefore, $\widehat{\gamma}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\Gamma_1')$. By the induction hypothesis we first conclude that γ is memory safe. Also, if γ terminates then there is a unique normal form γ' of γ, and $\widehat{\gamma'}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\Gamma_2')$. By the same reasons as above we have $\overline{\tau}(\Gamma_2) \equiv \overline{\tau}(\Gamma_2')$. Therefore, $\widehat{\gamma'}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\Gamma_2)$.

We are done with the soundness of the general purpose matching logic rules. We next prove the soundness of each of the KernelC language specific rules in Figure 5, recalling them but using exclusively the $\Downarrow$ notation for correctness pairs, i.e., desugaring the Hoare-like notation in some of the rules in Figure 5. The remaining proofs are very similar to each other, following the same structure: first pick some configuration γ matching the left pattern in the correctness pair which is the conclusion of the matching logic rule. Since computations are not modified by the user defined $\equiv$ equalities, it follows that γ has the same language construct in its computation as the matched pattern. Then, according to the evaluation strategy of the language construct under consideration, a number of configurations are obtained from γ, by rewriting parts of it using the rewrite logic semantics of KernelC, corresponding to the sub-expressions or sub-statements that need to be evaluated first. Then each of those is shown to match the left patterns in the correctness pairs in the hypothesis of the matching logic rule. By using the induction hypothesis then we conclude that all those computations are memory safe and, if they terminate, their normal forms match the right hand patterns. Then one can infer that γ is memory safe and, if it terminates, its unique normal form can be obtained by combining the normal forms of the other configurations. The fact that each of the intermediate configurations matches its right hand pattern implies that the combined normal form of γ also satisfies its right pattern.

$$\frac{\langle\langle K_1\rangle_k\, C\rangle \Downarrow \langle\langle\cdot\rangle_k\, C_1\rangle,\ \langle\langle K_2\rangle_k\, C_1\rangle \Downarrow \Gamma}{\langle\langle K_1\, K_2\rangle_k\, C\rangle \Downarrow \Gamma}$$

Suppose that the two correctness pairs above the line satisfy the property in Lemma 28 and let us show that the one below the line also satisfies it. Let τ be a ground $\mathit{free}(\langle\langle K_1\, K_2\rangle_k\, C\rangle)$-substitution and γ a configuration such that $\widehat{\gamma}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\langle\langle K_1\, K_2\rangle_k\, C\rangle)$, that is, such that $\widehat{\gamma}\,(\equiv\cup\Rrightarrow)^*\,\langle\langle\overline{\tau}(K_1)\,\overline{\tau}(K_2)\rangle_k\,\overline{\tau}(C)\rangle$. Since applications of $\equiv$ do not modify the computation structure, it follows that γ must be a configuration of the form $\langle\langle\overline{\tau}(K_1)\,\overline{\tau}(K_2)\rangle_k\, C_0\rangle$. Let γ_1 be the configuration $\langle\langle\overline{\tau}(K_1)\rangle_k\, C_0\rangle$. Again, since applications of $\equiv$ do not modify the computation structure, we can easily see that K_2 plays no role in showing that $\widehat{\gamma}\,(\equiv\cup\Rrightarrow)^*\,\langle\langle\overline{\tau}(K_1)\,\overline{\tau}(K_2)\rangle_k\,\overline{\tau}(C)\rangle$. Therefore, we can infer that $\widehat{\gamma_1}\,(\equiv\cup\Rrightarrow)^*\,\langle\langle\overline{\tau}(K_1)\rangle_k\,\overline{\tau}(C)\rangle$, that is, that $\widehat{\gamma_1}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\langle\langle K_1\rangle_k\, C\rangle)$. By the induction hypothesis it first follows that γ_1 is memory safe, and second that if γ_1 terminates then it has a unique normal form γ_1' and $\widehat{\gamma_1'}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\langle\langle\cdot\rangle_k\, C_1\rangle)$, that is, $\widehat{\gamma_1'}\,(\equiv\cup\Rrightarrow)^*\,\langle\langle\cdot\rangle_k\,\overline{\tau}(C_1)\rangle$. By similar arguments to the above, γ_1' must have the form $\langle\langle\cdot\rangle_k\, C_0'\rangle$. Let γ_2 be the configuration $\langle\langle\overline{\tau}(K_2)\rangle_k\, C_0'\rangle$. By similar arguments, it follows that $\widehat{\gamma_2}\,(\equiv\cup\Rrightarrow)^*\,\langle\langle\overline{\tau}(K_2)\rangle_k\,\overline{\tau}(C_1)\rangle$, that is, that $\widehat{\gamma_2}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\langle\langle K_2\rangle_k\, C_1\rangle)$. By the induction hypothesis it first follows that γ_2 is memory safe, and second that if γ_2 terminates then it has a unique normal form γ_2' and $\widehat{\gamma_2'}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\Gamma)$.

Since the rewrite rules in the rewrite logic semantics of KernelC only match and modify the computation structure at its top, there is only one way to rewrite γ in KernelC: first rewrite it the same way as γ_1 is rewritten, keeping the $\overline{\tau}(K_2)$

untouched at the bottom of the intermediate computations. If γ_1 does not terminate then γ does not terminate either and the latter is also memory safe, just like γ_1 is. If γ_1 terminates, then γ rewrites to γ_2 after processing all the steps corresponding to rewriting γ_1. From here on, the rewriting process is identical to γ_2's, so we conclude that γ is indeed memory safe. If γ_2 terminates, then γ terminates as well and, obviously, with the same normal form, γ'_2, which has the desired property that $\widehat{\gamma'_2}\,(\equiv\cup\Rrightarrow)^*\,\overline{\tau}(\Gamma)$.

$$\frac{\langle\langle K_1\rangle_k\, C\rangle \Downarrow \langle\langle I_1\rangle_k\, C_1\rangle,\ \langle\langle K_2\rangle_k\, C_1\rangle \Downarrow \langle\langle I_2\rangle_k\, C_2\rangle}{\langle\langle K_1\ \texttt{op}\ K_2\rangle_k\, C\rangle \Downarrow \langle\langle I_1\ op_{Int}\ I_2\rangle_k\, C_2\rangle}$$

The soundness proof of this rule is very similar to that of the rule above. Pick γ, γ_1 and γ_2 in the same manner as above. According to the structural equations of `op` in the rewriting logic semantics of KernelC, γ_1 is rewritten first (in a slightly and non-intrusively modified way, suffixing its computation with a frozen term), then γ_2, then the computations of their normal forms are summed. Since γ_1 and γ_2 match the left hand patterns in the rule hypothesis, they are memory safe and their normal forms, if they terminate, are I_1 and I_2, respectively. Thus we conclude that γ is also memory safe and its unique normal form, if it terminate, matches the right hand pattern in the conclusion of the rule.

$$\frac{\begin{array}{c}\langle\langle K_1\rangle_k\,\langle\varphi\rangle_{form}\, C\rangle \Downarrow \langle\langle I_1\rangle_k\,\langle\varphi_1\rangle_{form}\, C_1\rangle,\\ \langle\langle K_2\rangle_k\,\langle\varphi_1\wedge I_1\neq 0\rangle_{form}\, C_1\rangle \Downarrow \Gamma,\\ \langle\langle K_3\rangle_k\,\langle\varphi_1\wedge I_1=0\rangle_{form}\, C_1\rangle \Downarrow \Gamma\end{array}}{\langle\langle \texttt{if(}K_1\texttt{)}\ K_2\ \texttt{else}\ K_3\rangle_k\,\langle\varphi\rangle_{form}\, C\rangle \Downarrow \Gamma}$$

Let γ match the left hand pattern in the conclusion of the rule above, that is, there is some ground substitution τ such that $\widehat{\gamma}\,(\equiv\,\cup\,\Rrightarrow)^*\,\overline{\tau}(\langle\langle\texttt{if(}K_1\texttt{)}\ K_2\ \texttt{else}\ K_3\rangle_k\,\langle\varphi\rangle_{form}\, C\rangle)$. Let γ_1 be the configuration replacing the computation of γ by K_1. Then γ_1 matches the left hand pattern in the first correctness pair in the hypothesis of the rule, so γ_1 is memory safe. If γ_1 does not terminate, then γ will not terminate either, and so γ is also memory safe. Suppose that γ_1 terminates and let γ'_1 be its normal form. The computation of γ'_1 is either 0 or an integer number different from 0, the two cases being similar to analyze. Suppose it is different from 0 and let γ_2 be the configuration replacing the computation of γ' with K_2. Since γ'_1 matches the right pattern of the first hypothesis configuration pair, it follows that $I_1\neq 0$, so γ_2 matches the left hand pattern of the second hypothesis correctness pair. So γ_2 and γ are memory safe. If γ_2 terminates then so does γ and they have the same normal form.

$$\langle\langle X\rangle_k\,\langle X\mapsto I,\rho\rangle_{env}\, C\rangle \Downarrow \langle\langle I\rangle_k\,\langle X\mapsto I,\rho\rangle_{env}\, C\rangle$$

Straightforward, because this matching logic rule is identical to its corresponding rewriting logic rule in Figure 2.

$$\frac{\langle\langle K\rangle_k\, C\rangle \Downarrow \langle\langle I\rangle_k\,\langle\rho\rangle_{env}\, C'\rangle}{\langle\langle X\texttt{=}K\texttt{;}\rangle_k\, C\rangle \Downarrow \langle\langle\cdot\rangle_k\,\langle\rho[X\leftarrow I]\rangle_{env}\, C'\rangle}$$

As before, let γ match $\langle\langle X\texttt{=}K\texttt{;}\rangle_k\, C\rangle$ and let γ_1 change the computation cell $\langle X\texttt{=}K\texttt{;}\rangle_k$ by $\langle K\rangle_k$ in γ. Then γ_1 matches the pattern $\langle\langle K\rangle_k\, C\rangle$, so the induction hypothesis says that γ_1 is memory safe and, if it terminates, then its unique normal form γ'_1 matches $\langle\langle I\rangle_k\,\langle\rho\rangle_{env}\, C'\rangle$. Since the rewriting of γ in KernelC first processes K exactly as γ_1 does, if γ_1 does not terminate then γ does not terminate but is memory safe. If γ_1 terminates, then the only reduction left to terminate γ's rewriting is to assign the value I in the computation of γ'_1 to X. The resulting unique normal form of γ, say γ', then matches the pattern $\langle\langle\cdot\rangle_k\,\langle\rho[X\leftarrow I]\rangle_{env}\, C'\rangle$.

$$\frac{\langle\langle K\rangle_k\, C\rangle \Downarrow \langle\langle P\rangle_k\,\langle P\mapsto I\circledast\sigma\rangle_{mem}\, C'\rangle}{\langle\langle {*}K\rangle_k\, C\rangle \Downarrow \langle\langle I\rangle_k\,\langle P\mapsto I\circledast\sigma\rangle_{mem}\, C'\rangle}$$

Let γ match $\langle\langle {*}K\rangle_k\, C\rangle$ and let γ_1 replace $\langle {*}K\rangle_k$ by $\langle K\rangle_k$ in γ. Then γ_1 matches $\langle\langle K\rangle_k\, C\rangle$ and, by the induction hypothesis, γ_1 is memory safe and if it terminates then its unique normal form matches $\langle\langle P\rangle_k\,\langle P\mapsto I\circledast\sigma\rangle_{mem}\, C'\rangle$. If γ_1 does not terminate then γ does not terminate either but it is memory safe. If γ_1 terminates then let γ'_1 be its unique normal form. Since the rewriting of γ first processes K the same way γ_1 does, once K is processed in γ the resulting configuration is of the form $\langle\langle P \curvearrowright {*}\square\rangle_k\langle P\mapsto I\circledast\sigma\rangle_{mem}\, C'\rangle = \langle\langle {*}P\rangle_k\langle P\mapsto I\circledast\sigma\rangle_{mem}\, C'\rangle$. This configuration matches the lhs of the rewrite rule corresponding to pointer lookup in Figure 2, so it is rewritten to a configuration of the form $\langle\langle I\rangle_k\,\langle P\mapsto I\circledast\sigma\rangle_{mem}\, C'\rangle$, which is now a normal form. Therefore, γ is also memory safe when γ_1 terminates, and its unique normal form satisfies the desired pattern.

$$\frac{\begin{array}{c}\langle\langle K_1\rangle_k\, C\rangle \Downarrow \langle\langle P\rangle_k\, C_1\rangle,\\ \langle\langle K_2\rangle_k\, C_1\rangle \Downarrow \langle\langle I\rangle_k\,\langle P\mapsto I'\circledast\sigma\rangle_{mem}\, C_2\rangle\end{array}}{\langle\langle {*}K_1\texttt{=}K_2\texttt{;}\rangle_k\, C\rangle \Downarrow \langle\langle\cdot\rangle_k\,\langle P\mapsto I\circledast\sigma\rangle_{mem}\, C_2\rangle}$$

The soundness of the rule for pointer assignment above is similar to the ones for pointer lookup and normal assignment above, so we do not discuss it in detail.

$$\frac{\begin{array}{c}\langle\langle K_1\rangle_k\, C\rangle \Downarrow \langle\langle I\rangle_k\,\langle\varphi\rangle_{form}\, C_1\rangle,\\ \langle\langle K_2\rangle_k\,\langle\varphi\wedge I\neq 0\rangle_{form}\, C_1\rangle \Downarrow \langle\langle\cdot\rangle_k\, C\rangle\end{array}}{\langle\langle\texttt{while(}K_1\texttt{)}\ K_2\rangle_k\, C\rangle \Downarrow \langle\langle\cdot\rangle_k\,\langle\varphi\wedge I=0\rangle_{form}\, C_1\rangle}$$

Let γ match $\langle\langle\texttt{while(}K_1\texttt{)}\ K_2\rangle_k\, C\rangle$ and let γ_1 replace the computation cell $\langle\texttt{while(}K_1\texttt{)}\ K_2\rangle_k$ in γ by $\langle K_1\rangle_k$. Then γ_1 matches $\langle\langle K_1\rangle_k\, C\rangle$, so it is memory safe and, if it terminates, then its normal form γ'_1 matches $\langle\langle I\rangle_k\,\langle\varphi\rangle_{form}\, C_1\rangle$. If γ_1 does not terminate then γ does not terminate either, but γ is memory safe. Suppose that γ_1 terminates. If $I=0$ then, in the reduction of γ, the conditional in the semantics of `while` in Figure 2 takes the exit branch after processing K_1 and thus the normal form γ' of γ is γ'_1 with emptied computation, that is, with $\langle\cdot\rangle_k$ instead of $\langle I\rangle_k$. Then γ' matches the pattern $\langle\langle\cdot\rangle_k\,\langle\varphi\wedge I{=}0\rangle_{form}\,C_1\rangle$ and we are done with the case $I = 0$. If $I \neq 0$ then let γ_2 change the computation cell $\langle I\rangle_k$ in γ'_1 by $\langle K_2\rangle_k$. Then γ_2 matches the pattern $\langle\langle K_2\rangle_k\,\langle\varphi\wedge I{\neq}0\rangle_{form}\,C_1\rangle$, so by the induction hypothesis γ_2 is memory safe and, if it terminates, its normal form γ'_2 matches the pattern $\langle\langle\cdot\rangle_k\,C\rangle$. If γ_2 does not terminate then γ does not terminate either, but it is memory safe. Suppose that γ_2 terminates. Combining the rewrites of γ_1 and γ_2 and their unique normal forms satisfying the above-mentioned properties, we deduce that γ rewrites to a configuration δ which also matches the invariant pattern $\langle\langle\texttt{while(}K_1\texttt{)}\ K_2\rangle_k\,C\rangle$.

Iterating this process we get a finite or infinite sequence $\gamma = \delta_0, \delta_1, ...,$ such that for any $n \geq 0$ in the sequence the following holds: γ rewrites to δ_n and (a) δ_n does not terminate but is memory safe, or (b) δ_n is memory safe and terminates in a configuration satisfying the pattern $\langle\langle\cdot\rangle_k\,\langle\varphi\wedge I{=}0\rangle_{form}\,C_1\rangle$, or (c) δ_n rewrites to the next configuration in the sequence, δ_{n+1}, in a unique way. From this, we conclude that if γ has a normal form γ', then γ' indeed matches the pattern $\langle\langle\cdot\rangle_k\,\langle\varphi\wedge I{=}0\rangle_{form}\,C_1\rangle$.

$$\frac{\langle\langle K\rangle_k\,C\rangle \ \Downarrow\ \langle\langle N\rangle_k\,\langle\sigma\rangle_{mem}\,\langle\pi\rangle_{ptr}\,\langle\varphi\rangle_{form}\,\langle V\rangle_{bnd}\,C'\rangle}{\begin{array}{l}\langle\langle\texttt{malloc(}K\texttt{)}\rangle_k\,C\rangle \Downarrow \\ \qquad\langle\langle P\rangle_k\,\langle\sigma\circledast\sigma'\rangle_{mem}\,\langle\pi[P\leftarrow N]\rangle_{ptr}\,\langle\varphi\wedge\psi\rangle_{form}\,\langle V,P\rangle_{bnd}\,C'\rangle\end{array}}$$
$$(\text{where } \psi \text{ is } Dom(\sigma') = \overline{P, P+N-1})$$

Let γ match the pattern $\langle\langle\texttt{malloc(}K\texttt{)}\rangle_k\,C\rangle$ and let γ_1 be the configuration replacing the computation of γ by K. Since γ_1 matches $\langle\langle K\rangle_k\,C\rangle$, by the induction hypothesis γ_1 is memory safe and, if it terminates, its normal form γ'_1 matches $\langle\langle N\rangle_k\,\langle\sigma\rangle_{mem}\,\langle\pi\rangle_{ptr}\,\langle\varphi\rangle_{form}\,\langle V\rangle_{bnd}\,C'\rangle$. If γ_1 does not terminate we are done. If γ_1 terminates, then γ rewrites to a term of the form $\langle\langle P\rangle_k\,\langle\sigma\circledast\sigma'\rangle_{mem}\,\langle\pi[P\leftarrow N]\rangle_{ptr}\,C'\rangle$ for some concrete $P \in Nat$ and map σ' with $Dom(\sigma') = \overline{P, P+N-1}$, which therefore matches the desired pattern. Note that, since P is added as a bound variable in the desired pattern, it can match any concrete natural number; also, no new free variables are added to the pattern.

$$\frac{\langle\langle K\rangle_k\,C'\rangle \ \Downarrow\ \langle\langle P\rangle_k\,\langle\sigma\circledast\sigma'\rangle_{mem}\,\langle P\mapsto N,\pi\rangle_{ptr}\,\langle\varphi\wedge\psi\rangle_{form}\,C\rangle}{\langle\langle\texttt{free(}K\texttt{)}\rangle_k\,C'\rangle \ \Downarrow\ \langle\langle\cdot\rangle_k\,\langle\sigma\rangle_{mem}\,\langle\pi\rangle_{ptr}\,\langle\varphi\rangle_{form}\,C\rangle}$$
$$(\text{where } \psi \text{ is } Dom(\sigma') = \overline{P, P+N-1})$$

Let γ match the pattern $\langle\langle\texttt{free(}K\texttt{)}\rangle_k\,C'\rangle$ and let γ_1 be the configuration replacing the computation of γ by K. Since γ_1 matches $\langle\langle K\rangle_k\,C'\rangle$, by the induction hypothesis γ_1 is memory safe and, if it terminates, its normal form γ'_1 matches $\langle\langle P\rangle_k\,\langle\sigma\circledast\sigma'\rangle_{mem}\,\langle P\mapsto N,\pi\rangle_{ptr}\,\langle\varphi\wedge\psi\rangle_{form}\,C\rangle$. If γ_1 does not terminate we are done. If γ_1 terminates, then γ also has a unique well-terminated normal form which matches the pattern $\langle\langle\cdot\rangle_k\,\langle\sigma\rangle_{mem}\,\langle\pi\rangle_{ptr}\,\langle\varphi\rangle_{form}\,C\rangle$. □

5.7 Discussion

Matching logic, unlike Hoare logics, does *not* encode the entire program state in a formula. The rationale is that by keeping the configuration structure unaltered, one can more easily relate it to the actual program state and identify, by means of pattern matching, mathematical objects to reason about. However, for languages with simple configurations containing only an environment, for which Hoare logic was initially introduced [7], one can show that the Hoare and the matching logic formal systems are ultimately equivalent, that is, each can prove its rules from the other's.

Both separation logic [15, 8] and matching logic build upon the observation that program verification based on Hoare logic with FOL formulae frequently leads to complex and artificial encodings and proofs, particularly in the context of a heap. Separation logic considers the heap as *the* problem and extends FOL by assuming the heap at the core of its models; by introducing specialized logical connectives, like separating conjunction, it allows to state properties that hold in separate portions of the heap. The program state is still encoded as a formula like in Hoare logic, but using a more powerful logic. In matching logic, the heap plays *no special role*, being an algebraic data-type like the rest of the configuration. Since the heap is a map data-type, which is a set (of pairs), which is an associative (A) and commutative (C) binary operation, *matching modulo AC subsumes heap separation*: by definition, if a heap h matches a pattern $\sigma_1\circledast\sigma_2$ then it can be split in two disjoint sub-heaps h_1 and h_2 such that h_1 matches σ_1 and h_2 matches σ_2.

The above being said, matching logic's aim is neither to subsume nor to extend Hoare or separation logics. Its major aim is to provide a formal reasoning framework built upon the now well-understood and efficiently implemented operation of matching. By abstracting program states into configuration patterns expressed using the same formalism as the actual concrete configurations instead of logical formulae, matching logic is more executable in nature than the other logics, which comes with two important benefits: first, soundness results like Theorem 27 follow relatively easily; second and more importantly, as shown in Section 6, one can derive also relatively easily sound matching logic program verifiers directly from the executable semantics.

Techniques for soundly defining configuration constructs (like lists, trees, graphs, etc.) need to be developed. We hope that separation logic [15, 8] or shape analysis [18] predicate definitions may be adapted to our framework.

It is worth mentioning that configuration patterns have, in some sense, everything they need to be organized as a logic. Indeed, a pattern can be regarded as a kind of formula quantified existentially over its bound variables, with concrete configurations as models and with abstraction $\Rrightarrow$ as special implication connective. Then matching would become a decision procedure in such a setting, and explicit existential pattern variable Skolemization steps would be needed. However, without a practical need for more connectives and nesting for patterns with formulae, we think that organizing patterns in a logic is, at this moment, no more than a theoretical exercise.

6 Matching Logic Verification in K

Section 5 showed that a matching logic formal system for a language can be relatively automatically derived from a K definition of the language. That formal system can be used to prove properties about programs in many different ways, with a slight preference for a "forwards" approach, that is, one in which programs are processed in the order of their execution. We think that the forwards approach is not enforced and that one can develop weakest-precondition proving techniques based on matching logic, but we do not do it here. Here we show how one can derive a "forwards" matching logic program verifier based on the original K definition of the language. The program verifier is systematically derived, to such an extent that we believe that this process can actually be automated, same as the derivation of matching logic formal systems from K definitions. Unlike the matching logic formal system, the derived program verifier imposes a particular way to search for the matching logic proof, namely "forwards". Like in matching logic, the verifier will also work with configuration patterns. Like the K semantics of the language, the derived program verifier is also a formal K executable definition, but the purpose of its execution is to verify programs rather than execute them. The verifier will rewrite (configuration) patterns instead of (concrete) configurations. At branching points, e.g. `if`, it generates two rewrite tasks, one for each branch. To keep track of all the proof tasks, a top level bag structure is added, containing a soup of proof tasks.

Here are the steps necessary to transform a K language semantic definition in a matching logic program verifier for the defined language:

(1) ***Additional infrastructure.*** Since the verifier works with patterns like in matching logic, we first need to extend the syntax of configurations to patterns, like in matching logic. As expected, due to unavoidable undecidability aspects of program verification (even for memory safety: Proposition 5), the user needs to intervene in the verification process (e.g., by providing loop invariants). We provide one generic means for program annotations: *pattern assertions*. A pattern assertion is a pattern that one is free to state at any place in the computation corresponding to a program, including in the middle of an expression. Since the computation is implicit, to avoid clutter in pattern assertions we only mention the other cells of the pattern. The meaning of a pattern assertion is the expected one: when the program reaches that point during its execution, its configuration must *match* the asserted pattern. The simplest infrastructure needed to allow such pattern assertions is to simply extend the syntax of computations as follows:

$$K ::= \ldots \mid \mathsf{Bag}_{\cdot}^{--}[\mathit{CfgItem}]$$

In tools based on this verification approach one may want to prefix such pattern assertions with special keywords such as `assert`, or `invariant`, etc.

Another piece of infrastructure, useful to hold all the "proof tasks" in one place, is the following:

$$\mathit{Top} ::= \langle \mathsf{Bag}_{\cdot}^{--}[\mathit{Cfg}] \rangle_{\top}$$

Initially, the top bag will contain only one proof task, the original annotated program, but as shortly seen more proof tasks can be generated during a verification session.

(2) ***Assertion checking and cleanup.*** The verification process is going to proceed as follows: one starts with one "assumed" pattern in the top soup, which is regarded as a proof task. Then the rewriting engine "picks" a task from the soup and advances it one step, following mostly the existing rewrite semantics of each language construct as defined in K, with a few changes for some of the constructs, as shortly seen. When a pattern assertion is encountered, one has to prove that is can be matched by the current pattern. Therefore, we add the following rule:

$$\begin{array}{l}\langle\langle C \curvearrowright K\rangle_k\, C'\rangle \to \langle\langle K\rangle_k\, C\rangle \\ \quad \text{when } \langle\langle K\rangle_k\, C'\rangle\,(\equiv \cup \Rrightarrow)^{*}\langle\langle K\rangle_k\, C\rangle\end{array}$$

In other words, a particular proof task gets stuck on a pattern assertion is that cannot be shown to abstract the current pattern. For simplicity, in case of success we here choose to replace the current pattern by the abstract one; one can also keep the current pattern, as we do in our current prototype, because that contains more information than the abstract one. However, in that case one needs a special treatment for loop invariants, which we intend to discuss elsewhere.

The new rewrite system, as shortly seen, may yield more tasks in the soup. To clean them up, we add the following:

$$\langle\langle\cdot\rangle_k\, C\rangle \to \cdot$$

$$\langle\langle \mathit{false}\rangle_{\mathit{form}}\, C\rangle \to \cdot$$

The first rule above discards the completed tasks, and the second discards the infeasible ones. Completed tasks result when computations are completely processed, and infeasible ones when the original assumptions make some of the branches generated for proving infeasible.

(3) ***Make rules symbolic.*** One advantage of using rewriting in general and K in particular as a semantic language definitional framework is that rewrite rules make no difference between concrete and symbolic values, they all being terms. For example, the term $7 +_{Int} 3$ rewrites to 10, while the term $7 +_{Int} x +_{Int} 3$ rewrites to $10 +_{Int} x$ using the same rewriting machinery. This advantage makes it very convenient to build matching logic provers on top of K semantics because one needs not change most of the already existing semantics rules. Some changes, however, are needed. One needs to change all the rules which were conditional in the original K definition, because their conditions must now be "proved" using the pattern internal formula instead of just "checked" as in the original K definition. Also, language constructs which originally were defined by cases, like `if`, need now to generate one proof task for each of its cases. For example, here is the new rule for `if`, which replaces the two rules in the K definition in Figure 2:

$$\begin{array}{l}\langle\langle(\texttt{if}\,(I)\ K_2\ \texttt{else}\ K_3) \curvearrowright K\rangle_k\ \langle\varphi\rangle_{form}\ C\rangle\\ \quad \rightarrow \langle\langle K_2 \curvearrowright K\rangle_k \langle\varphi \wedge I{\neq}0\rangle_{form}\ C\rangle\\ \qquad \langle\langle K_3 \curvearrowright K\rangle_k \langle\varphi \wedge I{=}0\rangle_{form}\ C\rangle\end{array}$$

Let us now consider the `while` loop. Recall that its original K semantics was the following:

$$\begin{array}{l}\texttt{while(}K_1\texttt{)}K_2\\ \rightarrow \texttt{if(}K_1\texttt{)\{}K_2\texttt{;while(}K_1\texttt{)}K_2\texttt{\}}\end{array}$$

Following a similar reasoning to what we used to (for the time being informally) derive the matching rule for `while`, we can derive the following symbolic rule:

$$\begin{array}{l}\langle\langle(\texttt{while(}K_1\texttt{)}K_2) \curvearrowright K\rangle_k\ C\rangle\\ \rightarrow \langle\langle\texttt{if(}K_1\texttt{)\{}K_2 \curvearrowright C\texttt{\}}\ \texttt{else}\ K\rangle_k\ C\rangle\end{array}$$

This rule elegantly captures the two cases of the semantics of `while`: (a) the pattern at the beginning of the loop, say the "invariant", must also hold at the end of the loop body when the condition holds, and (2) the pattern is refined with the false condition for the remaining computation.

There are two more rules that special treatment because they have side conditions, the ones for memory allocation and deallocation. They are both rewrite variants of their corresponding matching logic rules:

$$\begin{array}{l}\langle\texttt{malloc(}N\texttt{);} \curvearrowright K\rangle_k\ \langle\sigma\rangle_{mem}\ \langle\pi\rangle_{ptr}\ \langle\varphi\rangle_{form}\ \langle V\rangle_{bnd}\\ \rightarrow \langle P \curvearrowright K\rangle_k\ \langle\sigma \circledast \sigma'\rangle_{mem}\ \langle\pi[P \leftarrow N]\rangle_{ptr}\ \langle\varphi \wedge \psi\rangle_{form}\ \langle V, P\rangle_{bnd}\\ \hfill (\text{where } \psi \text{ is } Dom(\sigma') = \overline{P, P+N-1})\end{array}$$

$$\begin{array}{l}\langle\texttt{free(}P\texttt{);} \curvearrowright K\rangle_k\ \langle\sigma \circledast \sigma'\rangle_{mem}\ \langle P \mapsto N, \pi\rangle_{ptr}\ \langle\varphi \wedge \psi\rangle_{form}\\ \rightarrow \langle K\rangle_k\ \langle\sigma\rangle_{mem}\ \langle\pi\rangle_{ptr}\ \langle\varphi\rangle_{form}\\ \hfill (\text{where } \psi \text{ is } Dom(\sigma') = \overline{P, P+N-1})\end{array}$$

Figure 7 shows all the above steps necessary to derive a matching logic program verifier from the K definition of KernelC, as well as a rewriting sequence using it.

Theorem 29 ***(Soundness and completeness of K verifier w.r.t. matching logic)*** *The following hold, where for an annotated computation K' like in Figure 7, K'° is the computation obtained by removing all pattern assertions from K':*

1. ***(Soundness)*** *If $\langle\langle\langle K' \curvearrowright C'\rangle_k\ C\rangle\rangle_\top \rightarrow^* \langle\cdot\rangle_\top$ using the K definition in Figure 7, then $\langle\langle K'^{\circ}\rangle_k\ C\rangle \Downarrow \langle\langle\cdot\rangle_k\ C'\rangle$ is derivable using the matching logic system in Figure 5;*
2. ***(Completeness)*** *If $\langle\langle K\rangle_k\ C\rangle \Downarrow \langle\langle\cdot\rangle_k\ C'\rangle$ is derivable using the matching logic formal system in Figure 5, then there is some annotated computation K' such that $K'^{\circ} = K$ and $\langle\langle\langle K' \curvearrowright C'\rangle_k\ C\rangle\rangle_\top \rightarrow^* \langle\cdot\rangle_\top$ using the K rewriting logic definition in Figure 7.*

Appendix 7 shows our current implementation of a matching logic prover for KernelC using Maude, as well as examples defining complex patterns on configurations.

Acknowledgments. Most of the work presented in this report has been done while the first author was on a sabbatical leave at Microsoft Research (MSR) in Redmond, WA. We wish to express our thanks to the members of the verification group at MSR, who attended presentations on this material and provided us with valuable feedback. We would like to warmly thank Margus Veanes of MSR and Ralf Sasse of UIUC for comments on preliminary drafts of this work which helped us significantly improve it. Last but not least, we would like to also thank the members of the formal systems laboratory at UIUC, particularly to Traian Florin Serbanuta and Chucky Ellison, for manifested interest in the subject presented in this paper and anticipated thanks for their help with a reimplementation and extension of our current matching logic verifier in Appendix 7.

References

[1] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings POPL'90*, pages 81–94. ACM, 1990.

[2] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. RS RS-04-26, BRICS, DAIMI, Dept. of Computer Science, University of Aarhus, Denmark, November 2004.

[5] S. P. Harbison and G. L. Steele. *C: A Reference Manual (5th Edition)*. Prentice Hall, 2002.

[6] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT, 1984.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[8] S. S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

(0) Start with the executable K semantics of the language (Fig. 2)

(1) Define patterns, like in matching logic, and add:

K ::= ... | $\mathsf{Bag}_{\cdot}^{--}[\mathit{CfgItem}]$
Top ::= $\langle \mathsf{Bag}_{\cdot}^{--}[\mathit{Cfg}]\rangle_{\top}$

(2) Add next three rules (assertion checking and completion):

$\langle\langle C \curvearrowright K\rangle_k\, C'\rangle \to \langle\langle K\rangle_k\, C\rangle \qquad \text{when } \langle\langle K\rangle_k\, C'\rangle\, (\equiv \cup \Rrightarrow)^* \langle\langle K\rangle_k\, C\rangle$
$\langle\langle\cdot\rangle_k\, C\rangle \to \cdot$
$\langle\langle \mathit{false}\rangle_{\mathit{form}}\, C\rangle \to \cdot$

(3) Replace rules for `if`, `while`, `malloc`, `free` with:

$\langle\langle(\texttt{if}\,(I)\; K_2\; \texttt{else}\; K_3) \curvearrowright K\rangle_k\, \langle\varphi\rangle_{\mathit{form}}\, C\rangle$
$\to \langle\langle K_2 \curvearrowright K\rangle_k \langle\varphi \wedge I{\neq}0\rangle_{\mathit{form}}\, C\rangle\;\; \langle\langle K_3 \curvearrowright K\rangle_k \langle\varphi \wedge I{=}0\rangle_{\mathit{form}}\, C\rangle$

$\langle\langle(\texttt{while}(K_1)K_2) \curvearrowright K\rangle_k\, C\rangle \to \langle\langle \texttt{if}(K_1)\{K_2 \curvearrowright C\}\,\texttt{else}\; K\rangle_k\, C\rangle$

$\langle \texttt{malloc}(N)\texttt{;} \curvearrowright K\rangle_k\, \langle\sigma\rangle_{\mathit{mem}}\, \langle\pi\rangle_{\mathit{ptr}}\, \langle\varphi\rangle_{\mathit{form}}\, \langle V\rangle_{\mathit{bnd}}$
$\to \langle P \curvearrowright K\rangle_k\, \langle\sigma \circledast \sigma'\rangle_{\mathit{mem}}\, \langle\pi[P \leftarrow N]\rangle_{\mathit{ptr}}\, \langle\varphi \wedge \psi\rangle_{\mathit{form}}\, \langle V, P\rangle_{\mathit{bnd}}$
(where ψ is $\mathit{Dom}(\sigma') = \overline{P, P+N-1}$)

$\langle \texttt{free}(P)\texttt{;} \curvearrowright K\rangle_k\, \langle\sigma \circledast \sigma'\rangle_{\mathit{mem}}\, \langle P \mapsto N, \pi\rangle_{\mathit{ptr}}\, \langle\varphi \wedge \psi\rangle_{\mathit{form}}$
$\to \langle K\rangle_k\, \langle\sigma\rangle_{\mathit{mem}}\, \langle\pi\rangle_{\mathit{ptr}}\, \langle\varphi\rangle_{\mathit{form}}$
(where ψ is $\mathit{Dom}(\sigma') = \overline{P, P+N-1}$)

Then $\langle\Gamma_{\mathit{start}}\rangle_\top \to^* \langle\Gamma_1\rangle_\top \to \langle\Gamma_2\rangle_\top \to \langle\Gamma_3\rangle_\top \to \langle\Gamma_4\, \Gamma_6\rangle_\top \to^* \langle\Gamma_5\, \Gamma_6\rangle_\top \to^* \langle\cdot\rangle_\top$ is a rewrite proof of the sum of first p numbers program, noting that $\langle C_1\rangle \Rrightarrow[\theta(n){=}1]\, \langle C_{\mathit{inv}}\rangle$, $\langle C_5\rangle \Rrightarrow[\theta(n){=}n{+}1]\, \langle C_{\mathit{inv}}\rangle$, $\langle C_6\rangle \Rrightarrow[\theta(n){=}p{+}1]\, \langle C_{\mathit{post}}\rangle$, where

$\Gamma_{\mathit{start}} \equiv \langle\langle \texttt{s=0;n=1;}\, C_{\mathit{inv}}\, \texttt{while(n!=p+1)\{s=s+n;n=n+1;\}}\, C_{\mathit{post}}\rangle_k\, C_{\mathit{pre}}\rangle$
$C_{\mathit{pre}} \equiv \langle \mathrm{p} \mapsto p, \rho\rangle_{\mathit{env}}\, \langle p \geq 0 \wedge \varphi\rangle_{\mathit{form}}\, \langle Z\rangle_{\mathit{bnd}}\, c$
$C_{\mathit{post}} \equiv \langle \mathrm{p} \mapsto p,\, \mathrm{s} \mapsto p(p{+}1)/2,\, \mathrm{n} \mapsto p{+}1,\, \rho\backslash s\backslash n\rangle_{\mathit{env}}\, \langle p \geq 0 \wedge \varphi\rangle_{\mathit{form}}\, \langle Z\rangle_{\mathit{bnd}}\, c$
$C_{\mathit{inv}} \equiv \langle \mathrm{p} \mapsto p, \mathrm{s} \mapsto n(n{-}1)/2, \mathrm{n} \mapsto n, \rho\backslash s\backslash n\rangle_{\mathit{env}} \langle p \geq 0 \wedge n \leq p{+}1 \wedge \varphi\rangle_{\mathit{form}}\, \langle n, Z\rangle_{\mathit{bnd}}\, c$
$\Gamma_1 \equiv \langle\langle C_{\mathit{inv}}\, \texttt{while(n!=p+1)\{s=s+n; n=n+1;\}}\, C_{\mathit{post}}\rangle_k\, C_1\rangle$
$C_1 \equiv \langle \mathrm{p} \mapsto p,\, \mathrm{s} \mapsto 0,\, \mathrm{n} \mapsto 1,\, \rho\backslash s\backslash n\rangle_{\mathit{env}}\, \langle p \geq 0 \wedge \varphi\rangle_{\mathit{form}}\, \langle Z\rangle_{\mathit{bnd}}\, c$
$\Gamma_2 \equiv \langle\langle \texttt{while(n!=p+1)\{s=s+n; n=n+1;\}}\, C_{\mathit{post}}\rangle_k\, C_{\mathit{inv}}\rangle$
$\Gamma_3 \equiv \langle\langle \texttt{if (n!=p+1) \{s=s+n; n=n+1;}\, C_{\mathit{inv}}\texttt{\} else}\, C_{\mathit{post}}\rangle_k\, C_{\mathit{inv}}\rangle$
$\Gamma_4 \equiv \langle\langle \texttt{s=s+n; n=n+1;}\, C_{\mathit{inv}}\rangle_k C_4\rangle \qquad \Gamma_5 \equiv \langle\langle C_{\mathit{inv}}\rangle_k C_5\rangle \qquad \Gamma_6 \equiv \langle\langle C_{\mathit{post}}\rangle_k C_6\rangle$
$C_4 \equiv \langle \mathrm{p} \mapsto p, \mathrm{s} \mapsto n(n{-}1)/2, \mathrm{n} \mapsto n, \rho\backslash s\backslash n\rangle_{\mathit{env}} \langle p \geq 0 \wedge n{<}p{+}1 \wedge \varphi\rangle_{\mathit{form}}\, \langle n, Z\rangle_{\mathit{bnd}}\, c$
$C_5 \equiv \langle \mathrm{p} \mapsto p, \mathrm{s} \mapsto n(n{+}1)/2, \mathrm{n} \mapsto n{+}1, \rho\backslash s\backslash n\rangle_{\mathit{env}} \langle p \geq 0 \wedge n{<}p{+}1 \wedge \varphi\rangle_{\mathit{form}} \langle n, Z\rangle_{\mathit{bnd}} c$
$C_6 \equiv \langle \mathrm{p} \mapsto p, \mathrm{s} \mapsto n(n{-}1)/2, \mathrm{n} \mapsto n, \rho\backslash s\backslash n\rangle_{\mathit{env}} \langle p \geq 0 \wedge n{=}p{+}1 \wedge \varphi\rangle_{\mathit{form}}\, \langle n, Z\rangle_{\mathit{bnd}}\, c$

Figure 7. Verifier in K and sample proof

[9] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[10] T. Lev-Ami and S. Sagiv. Tvla: A system for implementing static analyses. In *SAS*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301, 2000.

[11] J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[12] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theor. Computer Science*, 373(3):213–237, 2007.

[13] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[14] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[15] J. C. Reynolds. Separation logic: a logic for shared mutable data struct. In *Proceedings of LICS'02*, pages 55–74, 2002.

[16] G. Roşu. K: A Rewriting-Based Framework for Computations – Preliminary version. Technical Report UIUCDCS-R-2007-2926, University of Illinois, 2007.

[17] H. Rogers Jr. *Theory of Recursive Functions and Effective Computability*. MIT press, Cambridge, MA, 1987.

[18] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[19] T. F. Şerbănuţă, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Inf. and Comp.*, 2009. to appear; `http://dx.doi.org/10.1016/j.ic.2008.03.026`.

[20] M. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[21] M. Walicki and S. Meldal. Algebraic approaches to nondeterminism: An overview. *ACM Comput. Surv.*, 29(1):30–81, 1996.

[22] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

7. Matching Logic Verifier in Maude

The remaining of the paper shows our Maude implementation of a matching logic verifier for KernelC. We list the actual Maude code for the convenience of the interested reader, whom we assume already familiar with Maude [3]. We apologize for the lack of comments in the subsequent Maude code; this code is currently under major revision. We encourage the interested reader to contact us for the latest implementation and news.

```
in model-checker .


mod K is including INT .
  sorts Var NeVarList VarList K NeKList KList KConstant .
  subsorts Var Int KConstant < K < NeKList .
  subsorts Var < NeVarList < VarList NeKList < KList .
  ops null dot : -> KConstant .
  op _~>_ : K K -> K [assoc id: dot format (d +nib! o -) prec 100] .
  op `(`) : -> KList .
  op _,_ : NeVarList VarList -> NeVarList [assoc id: () prec 105] .
  op _,_ : VarList NeVarList -> NeVarList [ditto] .
  op _,_ : VarList VarList -> VarList [ditto] .
  op _,_ : NeKList KList -> NeKList [ditto] .
  op _,_ : KList NeKList -> NeKList [ditto] .
  op _,_ : KList KList -> KList [ditto] .
  op heat_ : KList -> K [memo prec 0] .
  op cool_ : KList -> K [prec 0] .
  eq heat(null) = cool(null) .
  eq heat(dot) = dot .
 ceq heat(K ~> K') = heat(K) ~> heat(K') if K =/= dot /\ K' =/= dot .
  op [_|_] : KList KList -> K .
  var K K' : K .  var Kl Kl' : KList .  var NeKl : NeKList .
  eq heat(K,NeKl) = heat(K) ~> [NeKl | ()] .
  eq heat() = cool() .
  eq cool(K') ~> [(K,Kl) | Kl'] = heat(K) ~> [Kl | (Kl',K')] .
  eq cool(K') ~> [() | NeKl] = cool(NeKl,K') .
endm


mod HEAP is including K .
  sorts FreshLoc LocType LocTypeList Heap NeHeap HeapDefTerm HeapDefName .
---  sort HeapVar
  subsort FreshLoc < K .  subsort LocType < LocTypeList .  subsorts Var HeapDefTerm < NeHeap < Heap .
  op #`(_`) : FreshLoc -> NeHeap [prec 0 format(r d d d o)] .  --- holds the counter for fresh location names
  op l : Nat -> FreshLoc .
  op n_ : FreshLoc -> FreshLoc .
  var N M : Nat .
  eq n(l(N)) = l(N + 1) .
  op _,_ : LocTypeList LocTypeList -> LocTypeList [assoc] .
  op [_,_,_] : K LocType K -> NeHeap .
  op empty : -> Heap .
  op _**_ : NeHeap Heap -> NeHeap [assoc comm id: empty format(d nisssb! o d)] .
  op _**_ : Heap Heap -> Heap [ditto] .
  ops heat_ cool_ : Heap -> K [ditto] .
  op __ : HeapDefName KList -> HeapDefTerm [prec 0] .          --- the KList arguments are "in"
  op ___ : HeapDefName KList KList -> HeapDefTerm [prec 0] .   --- the first KList arguments are "out", second are "in"

  op _._ : HeapDefName LocType -> LocType [prec 0] .
  op _[_] : HeapDefName HeapDefName -> HeapDefName [prec 0]  .
  op heap : Heap -> K .
endm


mod LANG-SYNTAX is including HEAP .
  op _+_ : K K -> K [ditto] .
  op _=_ : K K -> K [prec 60] .
  op _==_ : K K -> K [prec 50] .
  op _!=_ : K K -> K [prec 50] .
  op !_ : K -> K [prec 0] .
  op *_ : K -> K [prec 0] .

  op _&&_ : K K -> K [gather(e E) prec 55] .
  op _||_ : K K -> K [gather(e E) prec 59] .
  op if`(_`)_ : K K -> K [prec 93] .
  op if`(_`)_else_ : K K K -> K [prec 95] .
  op _;_ : K K -> K [assoc prec 100 gather(E e)] .
  op _; : K -> K [prec 99] .
  op while`(_`)_ : K K -> K .
  op {_} : K -> K .
  op {} : -> K .

  var K K' K1 K2 C : K .  var H : Heap .  var X : Var .  var NeXl : NeVarList .
  eq (K != K') = !(K == K') .
  eq K1 && K2 = if (K1) K2 else 0 .
  eq K1 || K2 = if (K1) 1 else K2 .
  eq ! K = if (K) 0 else 1 .
  eq if (K) K' = if (K) K' else {} .
  eq K ; = K .
  op [_] : Heap -> K .  --- heap patterns regarded as "bool expressions" that can have side effects (bindings of heap variables)
  op //@‘inv_while‘(_‘)_ : K K K -> K [prec 95] .
  op stop : -> K .
  op undef : -> K .
--- assume and assert
  ops (//@‘assume_) (//@‘assert_) (//@‘access_) : K -> K [prec 90] .
--- old
  op old : K -> K .
--- alloc (only typed locations)
  op alloc_ : LocTypeList -> K [prec 0] .
--- free
  op free_ : K -> K [prec 0] .
---
--- some translation rules
---

  op heap? : K -> Bool .
  eq heap?([H]) = true .
  eq heap?(heap(H)) = true .
  eq heap?(K) = false [owise] .
endm


mod PRE-FORMULAE is
  sorts True NtPreFormula PreFormula .  subsort True NtPreFormula < PreFormula .
  op True : -> True .
  op False : -> NtPreFormula .
  op _/\_ : NtPreFormula PreFormula -> NtPreFormula [assoc comm id: True prec 55] .
  op _/\_ : PreFormula PreFormula -> PreFormula [ditto] .
  op _=>_ : PreFormula PreFormula -> NtPreFormula [prec 61 strat(1 2 0)] .
  op ~_ : PreFormula -> NtPreFormula [prec 0] .

  var PreF PreF1 PreF2 : PreFormula .  var NtPreF NtPreF1 NtPreF2 : NtPreFormula .

  eq False /\ NtPreF = False .
  eq NtPreF /\ NtPreF = NtPreF .
  eq ~ NtPreF /\ NtPreF = False .
  eq ~ ~ PreF = PreF .
  eq ~ True = (False).PreFormula .
  eq ~ False = (True).PreFormula .

--- Even though the path condition is a conjunction, we may need disjunction for some theories, like sets.
  op _\/_  : PreFormula PreFormula -> NtPreFormula [assoc comm prec 59] .
  eq True \/ PreF = True .  eq False \/ PreF = PreF  .  eq PreF \/ PreF = PreF .  eq ~ PreF \/ PreF = True .

---  eq ~(PreF1 \/ PreF2) = ~(PreF1) /\ ~(PreF2) .

---  eq NtPreF /\ (PreF1 \/ PreF2) = NtPreF /\ PreF1 \/ NtPreF /\ PreF2 .

--- axiomatizing implication
***  eq PreF => True = True .
---  eq PreF => PreF = True .
***  eq NtPreF /\ PreF => NtPreF = True .

--- the ceq below is a bad idea, slows down a LOT
--- ceq PreF /\ PreF1 => PreF2 = True if PreF1 => PreF2 = True .

--- one of the two below may be needed
---  eq PreF /\ PreF1 => PreF /\ PreF2 = PreF /\ PreF1 => PreF2 .

***  eq PreF => (NtPreF1 /\ NtPreF2) = (PreF => NtPreF1) /\ (PreF => NtPreF2) .
***  eq PreF => (NtPreF1 \/ NtPreF2) = (PreF => NtPreF1) \/ (PreF => NtPreF2) .

***  eq PreF /\ ~(PreF1 /\ PreF2) => False = (PreF => PreF1) /\ (PreF => PreF2) .
endm


mod EQ-THEORY is including PRE-FORMULAE + K .
  op _===_ : K K -> NtPreFormula [comm] .

  var K K1 K2 : K .  var I I’ : Int .  var NeKl1 NeKl2 : NeKList .

  eq K + K1 === K + K2 = K1 === K2 .
  eq (K === K) = True .
  eq I === I’ = if I == I’ then True else False fi .
  eq K1 + K2 === null = False .

  eq (K1,NeKl1) === (K2,NeKl2) = (K1 === K2) /\ (NeKl1 === NeKl2) .
```

```
--- transitivity
  var PreF : PreFormula .
***  eq PreF /\ (K1 === K2) /\ (K === K1) => (K === K2) = True .
  eq (K1 === K2) /\ (K === K1) /\ ~(K === K2) = False .
endm


mod SEQ-THEORY is including EQ-THEORY .
  op nil : -> KConstant .
  op oneElemSeq : K -> K .
  op _::_ : K K -> K [assoc prec 1] .
  ops head tail last pref reverse : K -> K .

  var S S' S1 S1' S2 S2' S3 S3' K K1 K2 : K .  var PreF : PreFormula .
  eq oneElemSeq(head(S)) :: tail(S) = S .
  eq K === head(S) /\ ~(oneElemSeq(K) :: tail(S) :: S1 === S2) = K === head(S) /\ ~(S :: S1 === S2) .
  eq K === last(S) /\ ~(pref(S) :: oneElemSeq(K) :: S1 === S2) = K === last(S) /\ ~(S :: S1 === S2) .
  eq K === last(S) /\ ~(pref(S) :: oneElemSeq(K) === S2) = K === last(S) /\ ~(S === S2) .

  eq S1 === S1' /\ ~(S1 :: S2 === S1' :: S2) = False .
  eq S === nil /\ S1 === S :: S2 = S === nil /\ S1 === S2 .
  eq S === nil /\ S1 === S2 :: S = S === nil /\ S1 === S2 .
  eq S1 === S /\ S === S1' /\ ~(S1 :: S2 === S1' :: S2) = False .
  eq S1 === S1' /\ S === S1 :: S2 /\ S === S3 /\ ~(S1' :: S2 === S3) = False .
  eq S1 === S1' /\ S === S1 :: S2 /\ ~(S === S1' :: S2) = False .
  eq S1 === nil /\ ~(S1 :: S2 === S2) = False .
  eq S1 === nil /\ ~(S2 :: S1 === S2) = False .

  eq head(S) === K /\ ~(S === oneElemSeq(K) :: tail(S)) = False .
  eq head(S) === K /\ tail(S) === nil = S === oneElemSeq(K) .

  eq last(S) === K /\ ~(S === pref(S) :: oneElemSeq(K)) = False .


  eq head(S) === K /\ ~(reverse(oneElemSeq(K) :: tail(S)) :: S1 === S2) = head(S) === K /\ ~(reverse(S) :: S1 === S2) .

  eq S === nil /\ ~(reverse(S) === nil) = False .
  eq S === oneElemSeq(K) /\ ~(reverse(S) === S') = S === oneElemSeq(K) /\ ~(S === S') .
  eq reverse(S) :: oneElemSeq(K) = reverse(oneElemSeq(K) :: S) .
  eq reverse(S) === reverse(S') = S === S' .
  eq S === nil /\ ~(S1 :: reverse(S) === S2) = S === nil /\ ~(S1 === S2) .
  eq S === nil /\ ~(reverse(S) :: S1 === S2) = S === nil /\ ~(S1 === S2) .
  eq S === nil /\ reverse(S) :: S1 === S2 = S === nil /\ S1 === S2 .


  eq S1 :: S === S2 :: S = S1 === S2 .
  eq S :: S1 === S :: S2 = S1 === S2 .
  eq nil :: S = S .
  eq S === nil /\ ~(S === oneElemSeq(K)) = S === nil .
  eq S === nil /\ S === oneElemSeq(K) = False .

  eq S === oneElemSeq(K) /\ ~(head(S) === K) = False .
  eq S === oneElemSeq(K) /\ ~(tail(S) === nil) = False .

  eq S === nil /\ ~(S :: S1 === S2) = S === nil /\ ~(S1 === S2) .
  eq oneElemSeq(K) === nil = False .

  eq S === oneElemSeq(K) /\ ~(oneElemSeq(head(S)) === S') = S === oneElemSeq(K) /\ ~(oneElemSeq(K) === S') .

  eq S1 === S2 /\ ~(S1 :: S1' === S2 :: S2') = S1 === S2 /\ ~(S1' === S2') .
  eq oneElemSeq(K1) === oneElemSeq(K2) = K1 === K2 .
endm


mod TREE-THEORY is including EQ-THEORY .
  op emptyTree : -> KConstant .
  op Tree : K K K -> K .
  ops Data Left Right : K -> K .
  op mirror : K -> K .
  var K K' S S' S1 S1' S2 S2' : K .
  eq mirror(S) === emptyTree = S === emptyTree .
  eq mirror(emptyTree) = emptyTree .
  eq ~(S === emptyTree) /\ ~(mirror(S) === S') = ~(S === emptyTree) /\ ~(Tree(Data(S),mirror(Right(S)),mirror(Left(S))) === S') .
  eq S === emptyTree /\ ~(Tree(K,mirror(S),S') === S2) = S === emptyTree /\ ~(Tree(K,emptyTree,S') === S2) .
  eq S === emptyTree /\ ~(Tree(K,S',mirror(S)) === S2) = S === emptyTree /\ ~(Tree(K,S',emptyTree) === S2) .
  eq Left(S) === emptyTree /\ Right(S) === emptyTree /\ ~(mirror(S) === S') = Left(S) === emptyTree /\ Right(S) === emptyTree /\ ~(S === S') .
  eq Data(S) === K /\ Left(S) === S1 /\ Right(S) === S2 /\ ~(Tree(K,S1,S2) === S') = Data(S) === K /\ Left(S) === S1 /\ Right(S) === S2 /\ ~(S === S') .
  eq Tree(K,S1,S2) === Tree(K',S1',S2') = K === K' /\ S1 === S1' /\ S2 === S2' .
```

```
endm


mod SET-THEORY is including EQ-THEORY .
  ops emptySet allSet : -> KConstant .
  op oneElemSet : K -> K .

  op _INTERSECT_ : K K -> K [assoc comm prec 10] .
  op _MINUS_ : K K -> K [prec 13] .
  op _UNION_ : K K -> K [assoc comm prec 12] .
  op _IN_ : K K -> NtPreFormula [prec 15] .

  var A B C : K .

***(
  op _DIFF_ : K K -> K [assoc comm prec 5] .
  op EMP?_ : K -> NtPreFormula .
  eq allSet INTERSECT A = A .
  eq emptySet INTERSECT A = emptySet .
  eq A INTERSECT A = A .
  eq emptySet DIFF A = A .
  eq A DIFF A = emptySet .
  eq A INTERSECT (B DIFF C) = (A INTERSECT B) DIFF (A INTERSECT C) .
  eq A UNION B = A DIFF B DIFF (A INTERSECT B) .
  eq A MINUS B = A DIFF (A INTERSECT B) .

  eq A IN B = (A DIFF (A INTERSECT B) === emptySet) .

---  eq EMP?(emptySet) = True .
---  eq EMP?(A) /\ EMP?(B) = EMP?(A UNION B) .

  eq S === emptySet /\ ~((S INTERSECT S1) DIFF S2 === S3) = S === emptySet /\ ~(S2 === S3) .
  eq S === emptySet /\  ((S INTERSECT S1) DIFF S2 === S3) = S === emptySet /\  (S2 === S3) .
  eq S === emptySet /\ (S INTERSECT S' === S1) = S === emptySet /\ (emptySet === S1) .
  eq S === emptySet /\ S DIFF S1 === S2 = S === emptySet /\ S1 === S2 .
  eq S === S1 DIFF (S1 INTERSECT S2) /\ ~(S' DIFF (S INTERSECT S2) === S3) = S === S1 DIFF (S1 INTERSECT S2) /\ ~(S' === S3) .

  eq S1 DIFF (S1 INTERSECT S2) === emptySet /\ S2 DIFF (S1 INTERSECT S2) === S /\ ~(S DIFF S1 === S3)
   = S1 DIFF (S1 INTERSECT S2) === emptySet /\ S2 DIFF (S1 INTERSECT S2) === S /\ ~(S2 === S3) .

  eq A === B = EMP?(A DIFF B) .

---  eq ~ EMP?(A) = ...
---  eq EMP?(A) \/ EMP?(B) = ...
***)

--- LEMMAS as AXIOMS
  var S S' S1 S1' S2 S2' S3 S3' : K .  var L K : K .  var PreF : PreFormula .

---  eq oneElemSet(L) IN S1 UNION S2 = (oneElemSet(L) IN S1) \/ (oneElemSet(L) IN S2) .
---  eq oneElemSet(L) IN oneElemSet(K) = L === K .

  eq S IN S = True .
  eq S IN S UNION S' = True .
  eq emptySet IN S = True .
  eq emptySet UNION S = S .
  eq emptySet INTERSECT S = emptySet .
  eq S UNION S = S .
  eq S MINUS S = emptySet .
  eq S MINUS S' IN S = True .
  eq S UNION S' MINUS S' IN S = True .
  eq (S1 MINUS S2) MINUS S3 = S1 MINUS (S2 UNION S3) .
  eq (S1 MINUS S2) IN (S1 UNION S3) = True .
  eq S1 UNION S MINUS S2 UNION S = S1 MINUS S2 UNION S .
---  eq S1 UNION S2 IN S = (S1 IN S) /\ (S2 IN S) .
  eq (S1 MINUS S1') UNION (S2 MINUS S2') IN S1 UNION S2 = True .
  eq S UNION (S' MINUS S) = S UNION S' .
  eq (S1 UNION S2) MINUS S = (S1 MINUS S) UNION (S2 MINUS S) .

  eq (S1 MINUS S2) UNION S3 IN S1 = S3 IN S1 .
  eq (S1 MINUS S2) UNION S3 IN (S1 UNION S1') = S3 IN (S1 UNION S1') .

  eq S UNION S1 IN S UNION S2 = S1 IN S UNION S2 .

  eq S === emptySet /\ ~(S MINUS S' === S1) = S === emptySet /\ ~(emptySet === S1) .
  eq S === emptySet /\ (S INTERSECT S' === S1) = S === emptySet /\ (emptySet === S1) .
  eq S === emptySet /\ (S UNION S' === S1) = S === emptySet /\ (S' === S1) .
  eq S === emptySet /\ S' IN S = S === emptySet /\ S' === emptySet .
  eq S === emptySet /\ ~((S MINUS S') UNION S1 === S2) = S === emptySet /\ ~(S1 === S2) .

  eq emptySet === oneElemSet(K) = False .

  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S === S1 UNION S3 UNION S') = S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S === S2 UNION S') .
  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S === S1 UNION S3) = S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S === S2) .
  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S === (S1' MINUS (S1 UNION S3 UNION S')) UNION S2')
   = S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S === (S1' MINUS (S2 UNION S')) UNION S2') .
  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S IN (S1 UNION S3)) = S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S IN S2) .
  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S IN (S1 UNION S3 UNION S')) = S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S IN S2 UNION S') .
  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~((S MINUS (S1 UNION S3 UNION S')) UNION S1' IN S2')
   = S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~((S MINUS (S2 UNION S')) UNION S1' IN S2') .
  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~((S MINUS (S1 UNION S3)) UNION S1' IN S2')
   = S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~((S MINUS S2) UNION S1' IN S2') .
  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S MINUS (S1 UNION S3) IN S2') = S1 IN S2 /\ S3 === S2 MINUS S1 /\ ~(S MINUS S2 IN S2') .
  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\  (S === S1 UNION S3 UNION S') = S1 IN S2 /\ S3 === S2 MINUS S1 /\  (S === S2 UNION S') .
  eq S1 IN S2 /\ S3 === S2 MINUS S1 /\  (S === S1 UNION S3) = S1 IN S2 /\ S3 === S2 MINUS S1 /\ (S === S2) .

  eq S1 IN S2 /\ ~((S1 MINUS (S2 UNION S3)) UNION S IN S') = S1 IN S2 /\ ~(S IN S') .
  eq S1 IN S2 /\ S2 MINUS S3 === S UNION S' /\ ~(S1 MINUS (S3 UNION S UNION S1') IN S') = False .

  eq S1 IN S2 /\ ~(S1 MINUS S IN S2 UNION S') = False .
  eq S1 IN S2 /\ ~(S1 IN S2 UNION S) = False .


  eq S1 MINUS S2 === S3 /\ ~(S1 MINUS S IN S2 UNION S3) = False .
  eq S1 IN S2 /\ ~(S === (S1 MINUS (S2 UNION S2')) UNION S3) = S1 IN S2 /\ ~(S === S3) .
  eq S1 IN S2 /\ ~(S === (S1 MINUS (S2 UNION S2'))) = S1 IN S2 /\ ~(S === emptySet) .

  eq S1 IN S2 /\ ~((S1 MINUS S) UNION S' IN S2 UNION S2') = S1 IN S2 /\ ~(S' IN S2 UNION S2') .

***(
  eq PreF => S1 UNION S2 IN S = (PreF => S1 IN S) /\ (PreF => S2 IN S) .
  eq PreF /\ S === emptySet => S MINUS S' === emptySet = True .
  eq PreF /\ (S1 === S2) => S1 MINUS S IN S2 = True .
  eq PreF /\ S1 IN S2 /\ S2 === S2' => S1 MINUS (S2' UNION S3) === emptySet = True .
  eq PreF /\ S IN S1 /\ S1 === S2 UNION S3 /\ S2 === S2' => S MINUS (S' UNION S3) IN S2' = True .
  eq PreF /\ S1 IN S2 /\ S2 === S3 => S1 MINUS (S3 UNION S) IN S' = True .
  eq PreF /\ (S1 IN S2) /\ (S2 === S2') => S1 IN S2' UNION S3 = True .
  eq PreF /\ S1 IN S2 /\ S2 === S3 UNION S3' => S1 MINUS (S1' UNION S3) IN S3' = True .

  eq (S1 === S2) /\ (S === S3 UNION S1) /\ (S === S') /\ ~(S' === S3 UNION S2) = False .
  eq (S === S1 UNION S2) /\ (S1 === S1') /\ (S2 === S2') /\ ~(S === S1' UNION S2') = False .
  eq S1 IN S2 /\ S2 === S2' /\ ~(S1 MINUS S1' IN S2' UNION S3) = False .
  eq (S === S1 UNION S2) /\ (S1 === S1') /\ (S2 === S2' UNION S3) /\ (S3 === S3') /\ ~(S === S1' UNION S2' UNION S3') = False .
  eq S1 IN S2 /\ S2 === S3 UNION S3' /\ ~(S1 MINUS (S1' UNION S3) IN S3') = False .
  eq S1 UNION S2 === emptySet = S1 === emptySet /\ S2 === emptySet .

  eq S2 IN S1 /\ S1 MINUS S2 === S3 = S1 === S2 UNION S3 .
***)
endm


mod LIFTING is including LANG-SYNTAX .
  including EQ-THEORY .
  including SEQ-THEORY .
  including TREE-THEORY .
  including SET-THEORY .

  op ^ : K -> NtPreFormula .

  var K1 K2 : K .  var Kl1 Kl2 : KList .  var X : Var .

--- basic stuff
  eq ^(1) = True .
  eq ^(0) = False .
--- EQ
  eq ^((K1,Kl1) == (K2,Kl2)) = K1 === K2 /\ ^(Kl1 == Kl2) .
  eq () == () = 1 .
--- SET
  op _in_ : K K -> K [prec 15] .
  eq ^(K1 in K2) = K1 IN K2 .
endm


mod SUBST is including K + HEAP .
  sorts Subst NeSubst .  subsort NeSubst < Subst .
  op _<-_ : Var K -> NeSubst [prec 1] .
  op empty : -> Subst .
  op _,_ : NeSubst Subst -> NeSubst [assoc comm id: empty] .
  op _,_ : Subst Subst -> Subst [ditto] .
var L : FreshLoc .  var X : Var .  var K K1 K2 : K .  var Kl : KList .  var NeKl1 NeKl2 : NeKList .  var Subst : Subst .
  eq (K1,NeKl1) <- (K2,NeKl2) = (K1 <- K2),(NeKl1 <- NeKl2) .
---  eq K <- K = empty .

  op _[_] : K Subst -> K [prec 5] .
  op _[_] : KList Subst -> KList [ditto] .

  eq Kl[empty] = Kl .
  eq (NeKl1,NeKl2)[Subst] = (NeKl1[Subst]),(NeKl2[Subst]) .
  eq ()[Subst] = () .

  eq X[S1:[Subst],X <- K,S2:[Subst]] = K .

  eq X[Subst] = X [owise] .
eq L[Subst] = L [owise] .
eq Kc:KConstant[Subst] = Kc:KConstant .
endm


mod ENV is including SUBST .
  sorts FreshVar BasicEnv ProperEnv HeapEnv Env .
  subsorts BasicEnv < ProperEnv HeapEnv < Env .  subsort FreshVar < Var .
  op empty : -> BasicEnv .
  op {_,_} : Var K -> ProperEnv [prec 0] .
  op __ : BasicEnv BasicEnv -> BasicEnv [assoc comm id: empty] .
  op __ : ProperEnv ProperEnv -> ProperEnv [ditto] .
  op __ : Env Env -> Env [ditto] .
  op _[_] : Env Subst -> Env .
  op _[_] : Env Var -> [K] [prec 1] .
  op aux : Env Env Var -> [K] .
  op <_,_> : Env K -> [K] .

  op $‘(_‘) : FreshVar -> ProperEnv [prec 0 format(r d d d o)] .
  op v : Nat -> FreshVar .
  op n_ : FreshVar -> FreshVar .

  var X X' : Var .  var ?X ?F : FreshVar .  var K K' : K .  var Env Env' : Env .

  eq n(v(N:Nat)) = v(N:Nat + 1) .

--- sannity check, just in case somethig is wrong somewhere :-)
--- one can remove this equation eventually, to make environment checking a bit faster
op WrongENV : Var K K -> [Env] [format (r! o)] .
eq {X,K} {X,K'} = if K == K' then {X,K} else WrongENV(X,K,K') fi .

  eq Env[A:NeSubst, B:NeSubst] = Env[A:NeSubst][B:NeSubst] .
  eq Env[empty] = Env .

  eq ({X',K'} Env)[X <- K] = if X == X' then {X,K} Env else {X',K'} (Env[X <- K]) fi .
  eq $(?F)[X <- K] = $(?F) {X,K} .

  eq (Env $(?F))[X] = aux(Env $(?F), Env, X) .
  eq aux(Env, {X',K'} Env', X) = if X == X' then < Env, K' > else aux(Env, Env', X) fi .
  eq aux(Env $(?F), empty, X) = if X :: FreshVar then < Env {X,?F} $(n ?F), ?F > else < Env $(?F), X > fi .

eq {?X,?F} {?F,K} = {?X,K} .
endm


mod VALIDITY-CHECKER is including LIFTING + SAT-SOLVER + SUBST .
--- lift PreFormulae into Formulae
  op ^ : PreFormula -> Formula .
  var PreF PreF1 PreF2 : PreFormula .  var NtPreF1 NtPreF2 : NtPreFormula .
  eq ^(True) = True .
  eq ^(False) = False .
  eq ^(NtPreF1 /\ NtPreF2) = ^(NtPreF1) /\ ^(NtPreF2) .
---  eq ^(PreF1 \/ PreF2) = ^(PreF1) \/ ^(PreF2) .
  eq ^(~ PreF) = ~ ^(PreF) .

  op VALID : PreFormula -> Bool .
  op UNSAT‘(_‘) : PreFormula -> Bool [format(nr! d no nr! o)] .
  eq VALID(True) = true .
  eq VALID(False) = false .
--- the simplification equations do all the work for now; if needed, uncomment the following eq
---  op FAILED : PreFormula -> Bool .
---  eq VALID(PreF) = if tautCheck(^(PreF)) then true else FAILED(PreF) fi [owise] .
```

```
eq VALID(PreF1 => PreF2) = UNSAT(PreF1 /\ ~(PreF2)) .

---eq UNSAT(True) = false .
eq UNSAT(False) = true .
---eq UNSAT(PreF) = not tautCheck(~ ^(PreF)) [owise] .

var X : Var .  var F : FreshLoc .  var Kc : KConstant .

---eq UNSAT(X === Kc /\ PreF) = UNSAT(PreF[X <- Kc]) .
---eq UNSAT(F === Kc /\ PreF) = UNSAT(PreF[F <- Kc]) .

---op _[_] : PreFormula Subst -> [PreFormula] .

endm


mod CONFIG is including HEAP + ENV + VALIDITY-CHECKER .
  sort Config .
  op empty : -> Config .
  op __ : Config Config -> Config [assoc comm id: empty] .
  op <heap>_</heap> : Heap -> Config [format (rn! nssso nr! o)] .
  op <k>_</k> : K -> Config [format (gn! no ng! o)] .
  op <env>_</env> : Env -> Config [format (cn! o c! o)] .
  op <sat>_</sat> : PreFormula -> Config [format (mn! o m! o)] .
endm


mod CONFIGS is including CONFIG .
  sort Configs .
  ops done delete : -> Configs .
  op __ : Configs Configs -> Configs [prec 110 strat (1 0) frozen(2)] .
  var Cfgs Cfgs' Cfgs'' : [Configs] .  var N M : Nat .  var Cfg : Config .
  eq (Cfgs Cfgs') Cfgs'' = Cfgs (Cfgs' Cfgs'') .
  op <config>_</config> : Config -> Configs [format (yn! o yn! n)] .

--- this appears to work so far; if needed, you can replace it with the commented ceq
  eq <config> <sat> False </sat> ?Cfg:[Config] </config> = delete .
--- ceq <config> <sat> F:PreFormula </sat> ?Cfg:[Config] </config> = delete if VALID(~ F:PreFormula) .

  sort Result .
  op (_feasible and_infeasible paths) : Nat Nat -> Result .
  op [|_|] : K -> Result .
  op [| _,_,_ |] : Nat Nat Configs -> Result .
  eq [| N, M, delete Cfgs |] = [| N, M + 1, Cfgs |] .
  eq [| N, M, <config> <k> dot </k> Cfg </config> Cfgs |] = [| N + 1, M, Cfgs |] .
  eq [| N, M, done |] = N feasible and M infeasible paths .
endm


mod MATCH is including CONFIG .
  sort PreFormula*Subst .
  op (_,_) : PreFormula Subst -> PreFormula*Subst [prec 80] .
  op _|-_:=_ : PreFormula Heap Heap -> [PreFormula*Subst] [format (n nr! no nr! no n)] .
  op _|-_:=_ : PreFormula*Subst Heap Heap -> [PreFormula*Subst] [format (n nr! no nr! no n)] .

  var H H' : Heap .  var Hdt : HeapDefTerm .  var F : FreshLoc .  var Cond Cond' : PreFormula .
  var Hdn : HeapDefName .  var Kl Kl1 Kl2 Kl' Kl'' : KList .  var Subst Subst' : Subst .
  var ?H ?X : FreshVar .  var X : Var .

  eq Cond |- H' := H = (Cond,empty) |- H' := H .

  eq (Cond,Subst) |- H' := H' ** #(F) = (Cond,Subst) .
  eq (Cond,Subst) |- ?H ** H' := H ** H' ** #(F) = (Cond,(Subst, ?H <- heap(H))) .

***(
 ceq (Cond,Subst) |- Hdn(Kl')(Kl1) ** H' := Hdn(Kl'')(Kl2) ** H
   = (Cond,(Subst, Kl' <- Kl'')) |- H'[Kl' <- Kl''] := H
    if VALID(Cond => Kl1 === Kl2) .
 ceq (Cond,(?S1,(K <- K'),?S2)) = (Cond,(?S1,?S2)) if VALID(Cond => K === K') .
***)

 ceq (Cond,Subst) |- Hdn(Kl')(Kl1) ** H' := Hdn(Kl'')(Kl2) ** H
   = (Cond,(Subst,checkAndMakeSubst(Cond, Subst, Kl', Kl''))) |- H'[checkAndMakeSubst(Cond, Subst, Kl', Kl'')] := H
    if VALID(Cond => Kl1 === Kl2) .

  op checkAndMakeSubst : PreFormula Subst KList KList -> [Subst] .
  op _?_ : Bool [Subst] -> [Subst] .
  eq true ? S:[Subst] = S:[Subst] .
```

```
---  eq checkAndMakeSubst(Cond,(X <- K,Subst),(X,Kl'),(K'',Kl'')) = checkAndMakeSubst(Cond,(X <- K,Subst),(K,Kl'),(K'',Kl'')) .
  eq checkAndMakeSubst(Cond,Subst,(?X,Kl'),(K'',Kl'')) = ?X <- K'', checkAndMakeSubst(Cond,Subst,(Kl')[?X <- K''],Kl'') .
 ceq checkAndMakeSubst(Cond,Subst,(K',Kl'),(K'',Kl'')) = VALID(Cond => K' === K'') ? checkAndMakeSubst(Cond,Subst,Kl',Kl'')
    if not(K' :: FreshVar) .
  eq checkAndMakeSubst(Cond,Subst,(),()) = empty .

  var ?S1 ?S2 : [Subst] .  var K K' K'' : K .

  var L V : K .  var T : LocType .  var NeH1 NeH2 : NeHeap .  var NeKl1 NeKl2 : NeKList .

  op _[_] : Heap Subst -> [Heap] [prec 5] .
  eq ((empty).Heap)[Subst] = empty .
  eq #(F)[Subst] = #(F) .
  eq [L,T,V][Subst] = [L,T,V] .
  eq (NeH1 ** NeH2)[Subst] = (NeH1[Subst]) ** (NeH2[Subst]) .
  eq Hdn(Kl)[Subst] = Hdn(Kl[Subst]) .
  eq (Hdn(Kl')(Kl))[Subst] = Hdn(Kl'[Subst])(Kl[Subst]) .
endm


mod STUFF is including LANG-SYNTAX + CONFIGS + MATCH .
  op _? : Bool -> K [prec 0] .  eq true ? = dot .

  op derive : Heap K -> K .
  op answer : Heap LocType K -> K .
  var L K K' : K .  var T : LocType .  var H : Heap .  var NeKl : NeKList .  var Cfg : Config .
  eq derive(([L,T,K] ** H), L) = answer(H,T,K) .

  op cases_ : KList -> K .
  eq <config> <k> cases(K,NeKl) ~> K' </k> Cfg </config>
   = <config> <k> K ~> K' </k> Cfg </config> <config> <k> cases(NeKl) ~> K' </k> Cfg </config> .
  eq cases(K) = K .
endm


mod LANG-SEMANTICS is including STUFF .
  var L N K K' K1 K2 E E1 E2 C C' V : K .  var Kl Kl' : KList .  var X : Var .  var ?X : FreshVar .  var I I' : Int .
  var Env Env' : Env .
  var H H' : Heap .  var NeH NeH' : NeHeap .  var HDef : HeapDefName .
  var Cfg Cfg' : Config .
  var Cond Cond' : PreFormula .


  var F : FreshLoc .  var T : LocType .  var Tl : LocTypeList .
  var Subst : Subst .  var Fv : FreshVar .

  var P : K .
  eq  [| P |] =  [| 0, 0, <config>
                            <k> heat(P) </k>
                            <heap> #(l(0)) </heap>
                            <env> $(v(0)) </env>
                            <sat> True </sat>
                          </config> done |] .

  eq {K} = K .
  eq {} = dot .

--- integers
  eq heat(I) = cool(I) .
--- fresh locations
  eq heat(F) = cool(F) .

--- variable lookup
  eq <k> heat(X) ~> K </k> <env> Env </env> = <k> Env[X] ~> K </k> .
  eq <k> < Env,V > ~> K </k> = <k> cool(V) ~> K </k> <env> Env </env> .

--- location lookup: keep the heap intact while deriving it for debugging
  op *[] : -> K .  eq heat(* E) = (heat(E) ~> *[]) .
  eq <heap> H </heap> <k> cool(L) ~> *[] ~> K </k> = <heap> H </heap> <k> derive(H,L) ~> * L ~> K </k> .
  eq <heap> H' </heap> <k> answer(H,T,V) ~> * L ~> K </k> = <heap> H ** [L,T,V] </heap> <k> cool(V) ~> K </k> .

--- variable assignment
  op _=[] : Var -> K .  eq heat(X = E) = (heat(E) ~> (X = [])) .  eq cool(E) ~> X = [] = cool(X = E) .
  eq <k> cool(X = E) ~> K </k> <env> Env </env> = <k> K </k> <env> Env[X <- E] </env> .

--- location assignment
  ops (*[]=_) (*_=[]) : K -> K [prec 0] .  eq heat((* L) = E) = (heat(L) ~> *`[`]=_(E)) .  eq cool(L) ~> *`[`]=_(E) = (heat(E) ~> *_=`[`](L)) .
  eq <heap> H </heap> <k> cool(E) ~> *_=`[`](L) ~> K </k> = <heap> H </heap> <k> derive(H,L) ~> (* L = E) ~> K </k> .
  eq <heap> H </heap> <k> answer(H',T,V) ~> (* L = E) ~> K </k> = <heap> H' ** [L,T,E] </heap> <k> K </k> .
```

```
--- addition
  ops ([]+_) (_+[]) : K -> K .  eq heat(E1 + E2) = (heat(E1) ~> [] + E2) .
  eq cool(E1) ~> [] + E2 = (heat(E2) ~> E1 + []) .  eq cool(E2) ~> E1 + [] = cool(E1 + E2) .

--- ==
  ops ([]==_) (_==[]) : K -> K .  eq heat(E1 == E2) = (heat(E1) ~> [] == E2) .
  eq cool(E1) ~> [] == E2 = (heat(E2) ~> E1 == []) .  eq cool(E2) ~> E1 == [] = cool(E1 == E2) .

--- conditional
  op if`([]`)_else_ : K K -> K .
  eq heat(if (E) E1 else E2) = (heat(E) ~> if ([]) E1 else E2) .
  eq <config> <k> cool(E) ~> if ([]) E1 else E2 ~> K </k> <sat> Cond </sat> Cfg </config>
   = <config> <k> heat(E1) ~> K </k> <sat> Cond /\ ^(E) </sat> Cfg </config>
     <config> <k> heat(E2) ~> K </k> <sat> Cond /\ ~ ^(E) </sat> Cfg </config> .

--- stop
  eq <k> heat(stop) ~> K </k> = <k> dot </k> .

--- sequential composition
  eq heat(K1 ; K2) = (heat(K1) ~> heat(K2)) .

--- heap heating and cooling
  ops ([]**_) (_**[]) : NeHeap -> K .
  eq heat(NeH ** NeH') = (heat(NeH) ~> [] ** NeH') .
  eq cool(H) ~> [] ** NeH' = (heat(NeH') ~> H ** []) .  eq cool(H') ~> H ** [] = cool(H ** H') .
  eq heat((empty).Heap) = cool((empty).Heap) .

  op _([]) : HeapDefName -> K .
  eq heat(HDef(Kl)) = (heat(Kl) ~> HDef([])) .
  eq cool(Kl) ~> HDef([]) = cool(HDef(Kl)) .
--- we also heat the out parameters, but only to make sure they are consistently replaced by fresh variables
  ops (__([])) (_([])_) : HeapDefName KList -> K .
  eq heat(HDef(Kl')(Kl)) = (heat(Kl) ~> HDef(Kl')([])) .
  eq cool(Kl) ~> HDef(Kl')([]) = (heat(Kl') ~> HDef([])(Kl)) .
  eq cool(Kl') ~> HDef([])(Kl) = cool(HDef(Kl')(Kl)) .

--- heap "bool expression"
  eq heat([H]) = heat(H) .
  eq cool(heap(H)) = cool(H) .

--- assume
  op Assume : PreFormula -> K .
  op freezeKassume : K Env -> K .
  eq <k> heat(//@ assume(C)) ~> K </k> <env> Env $(Fv) </env> = <k> heat(C) ~> freezeKassume(K,Env) </k> <env> Env $(Fv) </env> .
  eq <k> cool(C) ~> freezeKassume(K,Env) </k> <env> Env' $(Fv) </env> = <k> Assume(^(C)) ~> K </k> <env> Env $(Fv) </env> .
  eq <k> Assume(Cond') ~> K </k> <sat> Cond </sat> = <k> K </k> <sat> Cond /\ Cond' </sat> .
  eq <heap> H' ** #(F) </heap> <k> cool(H) ~> K ~> freezeKassume(K',Env) </k> <sat> Cond </sat>
---   = <heap> H  ** #(F) </heap> <k> cool(1) ~> K ~> freezeKassume(K',Env) </k> <sat> Cond /\ assumeCond(H) </sat> .
    = <heap> H  ** #(F) </heap> <k> assumeCond(H) ~> cool(1) ~> K ~> freezeKassume(K',Env) </k> <sat> Cond </sat> .

  op assumeCond : Heap -> K .
  eq assumeCond(X) = dot .
  eq assumeCond(#(F)) = dot .
  eq assumeCond(NeH ** NeH') = (assumeCond(NeH) ~> assumeCond(NeH')) .
  eq assumeCond([L,T,V]) = Assume(~(L === null)) .


--- assert --- the "&& C" may not be necessary --- do some experiments with it and without it
  op Assert : PreFormula -> K .
  op freezeKassert : K -> K .
  eq <k> heat(//@ assert(C)) ~> K </k> = <k> heat(C) ~> freezeKassert(K) </k> .
  eq <k> cool(C) ~> freezeKassert(K) </k> = <k> Assert(^(C)) ~> K </k> .
  eq <k> Assert(Cond') ~> K </k> <sat> Cond </sat> = <k> VALID(Cond => Cond') ? ~> K </k> <sat> Cond </sat> .
 crl <heap> H </heap> <k> cool(H') ~> K ~> freezeKassert(K') </k> <sat> Cond </sat> <env> Env </env>
  => <heap> H </heap> <k> cool(1) ~> K ~> freezeKassert(K') </k> <sat> Cond' </sat> <env> Env[Subst] </env>
    if (Cond |- H' := H) => (Cond',Subst) .

--- old
  eq <k> heat(old(E)) ~> K </k> = <k> cool(E) ~> K </k> .

--- &&&
  op _&&&_ : K K -> K [gather(e E) prec 56] .
  eq <k> heat(K1 &&& K2) ~> K ~> freezeKassert(K') </k> = <k> heat(K1 && K2) ~> K ~> freezeKassert(K') </k> .
 ceq <k> heat(K1 &&& K2) ~> K ~> freezeKassume(K',Env) </k> = <k> heat(K2 && K1) ~> K ~> freezeKassume(K',Env) </k> if heap?(K2) .

--- access
  op //@`access([]) : -> K .
  eq heat(//@ access(K)) = (heat(K) ~> //@ access([])) .  eq cool(K) ~> //@ access([]) = dot .
```

```
--- alloc
***( --- this assumes that alloc may also fail, returning null
  eq <config> <heap> H ** #(F) </heap> <k> heat(alloc(Tl)) ~> K </k> <sat> Cond </sat> Cfg </config>
   = <config> <heap> H ** [F,Tl,null] ** #(n(F)) </heap> <k> cool(F) ~> K </k> <sat> Cond /\ ~ (F === null) </sat> Cfg </config>
     <config> <heap> H ** #(F) </heap> <k> cool(null) ~> K </k> <sat> Cond </sat> Cfg </config> .
  eq [L, (T,Tl), V] = [L, T, V] ** [L + 1, Tl, V] .
***)
  eq <heap> H ** #(F) </heap> <k> heat(alloc(Tl)) ~> K </k> <sat> Cond </sat>
   = <heap> H ** [F,Tl,null] ** #(n(F)) </heap> <k> cool(F) ~> K </k> <sat> Cond /\ ~ (F === null) </sat> .
  eq [L, (T,Tl), V] = [L, T, V] ** [L + 1, Tl, V] .

--- free
  op free([]) : -> K .
  eq heat(free(K)) = (heat(K) ~> free([])) .  eq cool(L) ~> free([]) = cool(free(L)) .
  eq <heap> H ** [L,T,V] </heap> <k> cool(free(L)) ~> K </k> = <heap> H </heap> <k> K </k> .


--- while
  eq //@ inv K' while (C) K
   = (//@ assert(K') ; flush ; cleanEnv ; cleanHeap ; cleanSat ; //@ assume(K') ; (if (C) (K ; //@ assert(K') ; stop))) .

  ops flush cleanEnv clean?Env cleanHeap cleanSat : -> K .
  eq heat(flush) = flush .
  eq heat(cleanEnv) = cleanEnv .
  eq heat(clean?Env) = clean?Env .
  eq heat(cleanHeap) = cleanHeap .
  eq heat(cleanSat) = cleanSat .
  rl <k> flush ~> K </k> => <k> K </k> .
 ceq <k> cleanEnv ~> K </k> <env> {X,V} Env </env> = <k> cleanEnv ~> K </k> <env> Env </env> if not heap?(V) .
  eq <k> cleanEnv ~> K </k> <env> Env </env> = <k> K </k> <env> Env </env> [owise] .
 ceq <k> clean?Env ~> K </k> <env> {?X,V} Env </env> = <k> clean?Env ~> K </k> <env> Env </env> if not heap?(V) .
  eq <k> clean?Env ~> K </k> <env> Env </env> = <k> K </k> <env> Env </env> [owise] .
  eq <heap> H ** #(F) </heap> <k> cleanHeap ~> K </k> = <heap> #(F) </heap> <k> K </k> .
  eq <k> cleanSat ~> K </k> <sat> Cond </sat> = <k> K </k> <sat> True </sat> .


--- SEQ theory
  eq heat(nil) = cool(nil) .
  ops ([]::_) (_::[]) : K -> K .
  eq heat(E1 :: E2) = (heat(E1) ~> [] :: E2) .
  eq cool(E1) ~> [] :: E2 = (heat(E2) ~> E1 :: []) .
  eq cool(E2) ~> E1 :: [] = cool(E1 :: E2) .

  op head`([]`) : -> K .
  eq heat(head(E)) = (heat(E) ~> head([])) .
  eq cool(E) ~> head([]) = cool(head(E)) .

  op tail`([]`) : -> K .
  eq heat(tail(E)) = (heat(E) ~> tail([])) .
  eq cool(E) ~> tail([]) = cool(tail(E)) .

  op reverse`([]`) : -> K .
  eq heat(reverse(E)) = (heat(E) ~> reverse([])) .
  eq cool(E) ~> reverse([]) = cool(reverse(E)) .

  op oneElemSeq`([]`) : -> K .
  eq heat(oneElemSeq(E)) = (heat(E) ~> oneElemSeq([])) .
  eq cool(E) ~> oneElemSeq([]) = cool(oneElemSeq(E)) .

--- TREE theory
  eq heat(emptyTree) = cool(emptyTree) .
  op Tree`([]`) : -> K .
  eq heat(Tree(K,E1,E2)) = (heat(K,E1,E2) ~> Tree([])) .
  eq cool(K,E1,E2) ~> Tree([]) = cool(Tree(K,E1,E2)) .

  op mirror`([]`) : -> K .
  eq heat(mirror(K)) = (heat(K) ~> mirror([])) .
  eq cool(K) ~> mirror([]) = cool(mirror(K)) .

--- SET theory
  eq heat(emptySet) = cool(emptySet) .
--- UNION
  ops ([]UNION_) (_UNION[]) : K -> K .  eq heat(E1 UNION E2) = (heat(E1) ~> [] UNION E2) .
  eq cool(E1) ~> [] UNION E2 = (heat(E2) ~> E1 UNION []) .  eq cool(E2) ~> E1 UNION [] = cool(E1 UNION E2) .
***(
--- DIFF
  ops ([]DIFF_) (_DIFF[]) : K -> K .  eq heat(E1 DIFF E2) = (heat(E1) ~> [] DIFF E2) .
  eq cool(E1) ~> [] DIFF E2 = (heat(E2) ~> E1 DIFF []) .  eq cool(E2) ~> E1 DIFF [] = cool(E1 DIFF E2) .
--- INTERSECT
  ops ([]INTERSECT_) (_INTERSECT[]) : K -> K .  eq heat(E1 INTERSECT E2) = (heat(E1) ~> [] INTERSECT E2) .
  eq cool(E1) ~> [] INTERSECT E2 = (heat(E2) ~> E1 INTERSECT []) .  eq cool(E2) ~> E1 INTERSECT [] = cool(E1 INTERSECT E2) .
***)
--- in
  ops ([]in_) (_in[]) : K -> K .  eq heat(E1 in E2) = (heat(E1) ~> [] in E2) .
  eq cool(E1) ~> [] in E2 = (heat(E2) ~> E1 in []) .  eq cool(E2) ~> E1 in [] = cool(E1 in E2) .
endm



mod EXAMPLES is including LANG-SEMANTICS .
  ops pgm1 pgm2 : -> K .
  ops wrong addHead allocAndAddHead appendRec1 appendWhile1 appendRec2 appendWhile2 reverse1 reverse2 reverseWhile disposeList disposeListWhile : -> K .
  ops enqueue dequeue transferOwner1 transferOwner2 stealQueue : -> K .
  ops allocTree mirror treeToList1 treeToList2 treeToListWhile : -> K .
  ops a b c d e f g h i j k l m n o p q r s t u v x y z temp ret return result from to left right : -> Var .
  ops ?a ?b ?c ?d ?e ?f ?g ?h ?i ?j ?k ?l ?m ?n ?o ?p ?q ?r ?s ?t ?u ?v ?x ?y ?z : -> FreshVar .
  ops list node queue seq tree graph stack : -> HeapDefName .
  ops frame frame1 frame2 frame3 rest rest1 rest2 rest3 : -> Var .
  ops ?frame ?frame1 ?frame2 ?frame3 ?rest ?rest1 ?rest2 ?rest3 : -> FreshVar .
  ops data next head tail left right : -> LocType .
  op ?z : -> FreshVar .
  op Schorr-Waite-tree : -> K .
  op root : -> Var .

ops requires ensures1 ensures2 ensures3 ensures4 : -> K .

  ops cleanTree markedTree stackTree : -> HeapDefName .
  ops marked switch generic : -> LocType .

  var D1 D1' D2 D2' : K .

  var In In1 In2 Out Out1 Out2 : K .
  var Set1 Set2 : FreshLoc .

  var L L' L1 L1' L1'' L2 L2' L2'' L3 L3' L3'' V K : K .  var F F1 F2 F3 : FreshLoc .  var H H' : Heap .
  var Cond Cond' : PreFormula .  var HDef : HeapDefName .  var Subst : Subst .

ops sa sb sq sf st : -> Var .
ops ?sa ?sb ?sq ?sf : -> FreshVar .

ops tl tr : -> Var .
ops ?tl ?tr : -> FreshVar .


  eq pgm1 =
--- simple program to test the various paths
    //@ assume(a != null || b != null || c != null) ;
    if (a == null)
      if (b == null) {
        a = c ;
        b = c
      } else {
        a = b ;
        c = a
      }
    else {
      b = a ;
      c = a
    } ;
    //@ assert(a != null && b != null && c != null)
.

---endm
---rew [| pgm1 |] .
---q



eq pgm2 =
---     b = a ; c = a ; d = a ; e = a ; f = a ; g = a ; h = a ; i = a ; j = a ; k = a ; l = a ; m = a ; n = a ; o = a ; p = a ; q = a ; r = a ;
     if (a) {} else {} ; --- 01
     if (b) {} else {} ; --- 02
     if (c) {} else {} ; --- 03
     if (d) {} else {} ; --- 04
     if (e) {} else {} ; --- 05
     if (f) {} else {} ; --- 06
     if (g) {} else {} ; --- 07
     if (h) {} else {} ; --- 08
     if (i) {} else {} ; --- 09
     if (j) {} else {} ; --- 10
     if (k) {} else {} ; --- 11
     if (l) {} else {} ; --- 12
     if (m) {} else {} ; --- 13
     if (n) {} else {} ; --- 14
     if (o) {} else {} ; --- 15
     if (p) {} else {} ; --- 16 : 24sec, 22.4sec with memo for heat
     if (q) {} else {} ; --- 17 : 49sec, 45.7sec with memo for heat
     if (r) {} else {} ; --- 18 : 99sec, 94.0sec with memo for heat --- 262144 paths
     //@ assert(1) ;
.

---endm
---rew [| pgm2 |] .
---q



---------------
--- list(L) ---
---------------
---
--- [list(L) ** H] =  L == null && [H] || [[L, (V,L')] ** list(L') ** H]
---

  eq assumeCond(list(L)) = dot .
 ceq (Cond,Subst) |- H' := H ** list(L) = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .
 ceq (Cond,Subst) |- H' ** list(L) := H = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .

  eq derive((list(L) ** H ** #(F)), L)
   = (Assert(~(L === null)) ~> answer([L + 1, node . next, n F] ** list(n F) ** H ** #(n n F), node . data, F)) .

  eq derive((list(L) ** H ** #(F)), L + 1)
   = (Assert(~(L === null)) ~> answer([L, node . data, F] ** list(n F) ** H ** #(n n F), node . next, n F)) .

 crl (Cond,Subst) |- H' := H ** [L, node . data, V] ** [L1, node . next, L2] ** list(L')
  => (Cond,Subst) |- H' := H ** list(L)   if VALID(Cond => L1 === L + 1 /\ L' === L2) .

 crl (Cond,Subst) |- H' := H ** [L, node . data, V] ** [L1, node . next, L']
  => (Cond,Subst) |- H' := H ** list(L)   if VALID(Cond => L1 === L + 1 /\ L' === null) .


var S S1 S2 S3 : K .

------------------
--- list(S)(L) ---
------------------
---
--- [list(S)(L) ** H] =  L == null && S == nil && [H]
---                   || L != null && [[L, (V,L')] ** list(S')(L') ** H] && S == V :: S'
---
  eq assumeCond(list(S)(L)) = dot .
 ceq (Cond,Subst) |- H' := H ** list(S)(L) = (Cond /\ S === nil, Subst) |- H' := H         if VALID(Cond => L === null) .
 ceq (Cond,Subst) |- H' ** list(S)(L) := H
   = (Cond,(Subst, checkAndMakeSubst(Cond,Subst,S,nil))) |- H'[checkAndMakeSubst(Cond,Subst,S,nil)] := H if VALID(Cond => L === null) .

  eq derive((list(S)(L) ** H ** #(F)), L)
   = (Assert(~(L === null)) ~> Assume(~(S === nil) /\ head(S) === F)
     ~> answer([L + 1, node . next, n F] ** list(tail(S))(n F) ** H ** #(n n F), node . data, F)) .

  eq derive((list(S)(L) ** H ** #(F)), L + 1)
   = (Assert(~(L === null)) ~> Assume(~(S === nil) /\ head(S) === F)
     ~> answer([L, node . data, F] ** list(tail(S))(n F) ** H ** #(n n F), node . next, n F)) .

 crl (Cond,Subst) |- H' := H ** [L, node . data, V] ** [L1, node . next, L2] ** list(S)(L')
  => (Cond,Subst) |- H' := H ** list(oneElemSeq(V) :: S)(L)  if VALID(Cond => L1 === L + 1 /\ L' === L2) .

 crl (Cond,Subst) |- H' := H ** [L, node . data, V] ** [L1, node . next, L']
  => (Cond,Subst) |- H' := H ** list(oneElemSeq(V))(L)   if VALID(Cond => L1 === L + 1 /\ L' === null) .


---------------
--- node(L) ---
---------------
---
--- [node(L) ** H] = [[L, (V,L')] ** H]
---
```

```
  eq assumeCond(node(L)) = dot .
 ceq (Cond,Subst) |- H' := H ** node(L) = (Cond,Subst) |- H' := H   if VALID(Cond => L === null) .
 ceq (Cond,Subst) |- H' ** node(L) := H = (Cond,Subst) |- H' := H   if VALID(Cond => L === null) .

  eq derive(node(L) ** H ** #(F), L)
   = (Assert(~(L === null)) ~> answer([L + 1, node . next, n F] ** H ** #(n n F), node . data, F)) .

  eq derive(node(L) ** H ** #(F), L + 1)
   = (Assert(~(L === null)) ~> answer([L, node . data, F] ** H ** #(n n F), node . next, n F)) .

 crl (Cond,Subst) |- H' := H ** [L,node . data,V] ** [L1,node . next,L']
  => (Cond,Subst) |- H' := H ** node(L)   if VALID(Cond => L1 === L + 1) .


------------------
--- node(V)(L) ---
------------------
---
--- [node(V)(L) ** H] = [[L, (V,L')] ** H]
---
--- ceq (Cond,Subst) |- H' := H ** node(V)(L) = (Cond,Subst) |- H' := H   if VALID(Cond => L === null) .
--- ceq (Cond,Subst) |- H' ** node(V)(L) := H = (Cond,Subst) |- H' := H   if VALID(Cond => L === null) .

  eq assumeCond(node(V)(L)) = Assume(~(L === null)) .

  eq derive(node(V)(L) ** H ** #(F), L)
   = (Assert(~(L === null)) ~> answer([L + 1, node . next, F] ** H ** #(n F), node . data, V)) .

  eq derive(node(V)(L) ** H ** #(F), L + 1)
   = (Assert(~(L === null)) ~> answer([L, node . data, V] ** H ** #(n F), node . next, F)) .

 crl (Cond,Subst) |- H' := H ** [L,node . data,V] ** [L1,node . next,L']
  => (Cond /\ ~(L === null), Subst) |- H' := H ** node(V)(L)   if VALID(Cond => L1 === L + 1) .


----------------
--- queue(L) ---
----------------
---
--- [queue(L) ** H] = [[L, (Q,Q')] ** seq(L')(Q,Q') ** H]
---

  eq assumeCond(queue(L)) = dot .

 ceq (Cond,Subst) |- H' := H ** queue(L) = (Cond,Subst) |- H' := H   if VALID(Cond => L === null) .
 ceq (Cond,Subst) |- H' ** queue(L) := H = (Cond,Subst) |- H' := H   if VALID(Cond => L === null) .

  eq derive(queue(L) ** H ** #(F), L)
   = (Assert(~(L === null))
     ~> cases(answer([L + 1, queue . tail, null] ** H ** #(F), queue . head, null),
              Assume(~(F === null) /\ ~(n F === null))
              ~> answer([L + 1, queue . tail, n F] ** seq(n n F)(F,n F) ** H ** #(n n n F), queue . head, F))) .

  eq derive(queue(L) ** H ** #(F), L + 1)
   = (Assert(~(L === null))
     ~> cases(answer([L, queue . head, null] ** H ** #(F), queue . tail, null),
              Assume(~(F === null) /\ ~(n F === null))
              ~> answer([L, queue . head, F] ** seq(n n F)(F,n F) ** H ** #(n n n F), queue . tail, n F))) .

 crl (Cond,Subst) |- H' := H ** [L,queue . head,L'] ** [L1,queue . tail,L1']
  => (Cond,Subst) |- H' := H ** queue(L)
    if VALID(Cond => L1 === L + 1 /\ L' === null /\ L1' === null) .

 crl (Cond,Subst) |- H' := H ** [L,queue . head,L'] ** [L1,queue . tail,L1'] ** seq(L2)(L',L1')
  => (Cond,Subst) |- H' := H ** queue(L)   if VALID(Cond => L1 === L + 1) .



-------------------
--- queue(S)(L) ---
-------------------
  eq assumeCond(queue(S)(L)) = cases(Assume(S === nil), Assume(~(S === nil) /\ ~(L === null))) .

 ceq (Cond,Subst) |- H' := H ** queue(S)(L) = (Cond /\ S === nil,Subst) |- H' := H   if VALID(Cond => L === null) .
 ceq (Cond,Subst) |- H' ** queue(S)(L) := H
   = (Cond,(Subst,checkAndMakeSubst(Cond,Subst,S,nil))) |- H'[checkAndMakeSubst(Cond,Subst,S,nil)] := H  if VALID(Cond => L === null) .

  eq derive(queue(S)(L) ** H ** #(F), L)
   = (Assert(~(L === null))
     ~> cases(Assume(S === nil) ~> answer([L + 1, queue . tail, null] ** H ** #(F), queue . head, null),
              Assume(~(S === nil) /\ ~(F === null) /\ ~(n F === null))
              ~> answer([L + 1, queue . tail, n F] ** seq(S,n n F)(F,n F) ** H ** #(n n n F), queue . head, F))) .

  eq derive(queue(S)(L) ** H ** #(F), L + 1)
   = (Assert(~(L === null))
     ~> cases(Assume(S === nil) ~> answer([L, queue . head, null] ** H ** #(F), queue . tail, null),
              Assume(~(S === nil) /\ ~(F === null) /\ ~(n F === null))
              ~> answer([L, queue . head, F] ** seq(S,n n F)(F,n F) ** H ** #(n n n F), queue . tail, n F))) .

 crl (Cond,Subst) |- H' := H ** [L,queue . head,L'] ** [L1,queue . tail,L1']
  => (Cond,Subst) |- H' := H ** queue(nil)(L)
    if VALID(Cond => L1 === L + 1 /\ L' === null /\ L1' === null) .

 crl (Cond,Subst) |- H' := H ** [L,queue . head,L'] ** [L1,queue . tail,L1'] ** seq(S,L2)(L',L1')
  => (Cond,Subst) |- H' := H ** queue(S)(L)   if VALID(Cond => L1 === L + 1) .



---------------------
--- seq(L)(L1,L2) ---
---------------------

  eq assumeCond(seq(L)(L1,L2)) = Assume(~(L1 === null) /\ ~(L2 === null)) .

  eq derive(seq(L)(L1,L2) ** H ** #(F), L1)
   = cases(Assume(L1 === L2) ~> answer([L1 + 1,node . next,L] ** H ** #(n F), node . data, F),
          Assume(~(L1 === L2)) ~> answer([L1 + 1,node . next,n F] ** seq(L)(n F,L2) ** H ** #(n n F), node . data, F)) .

  eq derive(seq(L)(L1,L2) ** H ** #(F), L1 + 1)
   = cases(Assume(L1 === L2) ~> answer([L1,node . data,F] ** H ** #(n F), node . next, L),
          Assume(~(L1 === L2)) ~> answer([L1,node . data,F] ** seq(L)(n F,L2) ** H ** #(n n F), node . next, n F)) .

  eq derive(seq(L)(L1,L2) ** H ** #(F), L2 + 1)
   = cases(Assume(L1 === L2) ~> answer([L1,node . data,F] ** H ** #(F), node . next, L),
          Assume(~(L1 === L2)) ~> answer(seq(L2)(L1,n F) ** [L2,node . data,F] ** H ** #(n n F), node . next, L)) .

  crl (Cond,Subst) |- H' := H ** [L,node . data,V] ** [L1,node . next,L']
   => (Cond,Subst) |- H' := H ** seq(L')(L,L)   if VALID(Cond => L1 === L + 1) .

  crl (Cond,Subst) |- H' := H ** seq(L)(L1,L2) ** seq(L')(L1',L2')
   => (Cond,Subst) |- H' := H ** seq(L')(L1,L2')   if VALID(Cond => L1' === L) .

  crl (Cond,Subst) |- H' := H ** node(L) ** #(F)
   => (Cond,Subst) |- H' := H ** seq(F)(L,L) ** #(n F)   if VALID(Cond => ~(L === null)) .

--- something more general may replace the rule below; for now it is good enough
  crl (Cond,Subst) |- H' ** list(L) ** seq(L)(L1,L2) := H
   => (Cond,Subst) |- H' ** list(L1) := H   if VALID(Cond => ~(L1 === null)) .

  crl (Cond,Subst) |- H' := H ** list(L) ** seq(L)(L1,L2)
   => (Cond,Subst) |- H' := H ** list(L1)   if VALID(Cond => ~(L1 === null)) .


-----------------------
--- seq(S,L)(L1,L2) ---
-----------------------
  eq assumeCond(seq(S,L)(L1,L2)) = Assume(~(L1 === null) /\ ~(L2 === null) /\ ~(S === nil)) .

  eq derive(seq(S,L)(L1,L2) ** H ** #(F), L1)
   = (Assume(~(S === nil)) ~>
     cases(Assume(L1 === L2) ~> Assume(S === oneElemSeq(F)) ~> answer([L1 + 1,node . next,L] ** H ** #(n F), node . data, F),
           Assume(~(L1 === L2)) ~> Assume(head(S) === F)
           ~> answer([L1 + 1,node . next,n F] ** seq(tail(S),L)(n F,L2) ** H ** #(n n F), node . data, F))) .

  eq derive(seq(S,L)(L1,L2) ** H ** #(F), L1 + 1)
   = (Assume(~(S === nil)) ~>
     cases(Assume(L1 === L2) ~> Assume(S === oneElemSeq(F)) ~> answer([L1,node . data,F] ** H ** #(F), node . next, L),
           Assume(~(L1 === L2)) ~> Assume(head(S) === F)
           ~> answer([L1,node . data,F] ** seq(tail(S),L)(n F,L2) ** H ** #(n n F), node . next, n F))) .

  eq derive(seq(S,L)(L1,L2) ** H ** #(F), L2 + 1)
   = (Assume(~(S === nil)) ~>
     cases(Assume(L1 === L2) ~> Assume(S === oneElemSeq(F)) ~> answer([L1,node . data,F] ** H ** #(n F), node . next, L),
           Assume(~(L1 === L2)) ~> Assume(last(S) === F)
           ~> answer(seq(pref(S),L2)(L1,n F) ** [L2,node . data,F] ** H ** #(n n F), node . next, L))) .

  crl (Cond,Subst) |- H' := H ** [L,node . data,V] ** [L1,node . next,L']
   => (Cond,Subst) |- H' := H ** seq(oneElemSeq(V),L')(L,L)   if VALID(Cond => L1 === L + 1) .
  crl (Cond,Subst) |- H' := H ** seq(S1,L)(L1,L2) ** seq(S2,L')(L1',L2')
   => (Cond,Subst) |- H' := H ** seq(S1 :: S2, L')(L1,L2')   if VALID(Cond => L1' === L) .

---  crl (Cond,Subst) |- H' := H ** node(V)(L) ** #(F)
---   => (Cond,Subst) |- H' := H ** seq(oneElemSeq(V),F)(L,L) ** #(n F)   if VALID(Cond => ~(L === null)) .
   rl (Cond,Subst) |- H' := H ** node(V)(L) ** #(F) => (Cond /\ ~(L === null), Subst) |- H' := H ** seq(oneElemSeq(V),F)(L,L) ** #(n F) .

--- something more general may replace the rule below; for now it is good enough
---  crl (Cond,Subst) |- H' := H ** list(S)(L) ** #(F)
---   => (Cond,Subst) |- H' := H ** seq(head(S),F)(L,L) ** list(tail(S))(F) ** #(n F)   if VALID(Cond => ~(L === null)) .

  crl (Cond,Subst) |- H' ** seq(S1,L1)(L,L) := H ** list(S)(L) ** #(F)
   => (Cond,(Subst,checkAndMakeSubst(Cond,Subst,(S1,L1),(oneElemSeq(head(S)),F))))
      |- H'[checkAndMakeSubst(Cond,Subst,(S1,L1),(oneElemSeq(head(S)),F))] := H ** list(tail(S))(F) ** #(n F)
     if VALID(Cond => ~(L === null)) .

   rl (Cond,Subst) |- H' ** list(S)(L) := H ** seq(S1,L1)(L,L) ** #(F)
   => (Cond,(Subst,checkAndMakeSubst(Cond,Subst,(S),(S1 :: F))))
      |- H'[checkAndMakeSubst(Cond,Subst,(S),(S1 :: F))] ** list(F)(L1) := H ** #(n F) .

   rl (Cond,Subst) |- H' ** seq(S,L)(L1,L3) := H ** seq(S1,L3)(L1,L2) ** list(S3)(L3) ** #(F)
   => (Cond,(Subst,checkAndMakeSubst(Cond,Subst,(S,L),((S1 :: oneElemSeq(head(S3))),F))))
      |- H'[checkAndMakeSubst(Cond,Subst,(S,L),((S1 :: oneElemSeq(head(S3))),F))] := H ** list(tail(S3))(F) ** #(n F) .

***(
  crl (Cond,Subst) |- H' ** list(S2)(L) ** seq(S1,L)(L1,L2) := H
   => (Cond,Subst) |- H' ** list(S1 :: S2)(L1) := H   if VALID(Cond => ~(L1 === null)) .
***)

  crl (Cond,Subst) |- H' := list(S2)(L) ** seq(S1,L)(L1,L2) ** H
   => (Cond,Subst) |- H' := list(S1 :: S2)(L1) ** H   if VALID(Cond => ~(L1 === null)) .



---------------
--- tree(L) ---
---------------
  eq assumeCond(tree(L)) = dot .

 ceq (Cond,Subst) |- H' := H ** tree(L) = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .
 ceq (Cond,Subst) |- H' ** tree(L) := H = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .

  eq derive((tree(L) ** H ** #(F)), L)
   = (Assert(~(L === null))
      ~> answer([L + 1, tree . left, n F] ** tree(n F) ** [L + 2, tree . right, n n F] ** tree(n n F) ** H ** #(n n n F), tree . data, F)) .

  eq derive((tree(L) ** H ** #(F)), L + 1)
   = (Assert(~(L === null))
      ~> answer([L, tree . data, F] ** tree(n F) ** [L + 2, tree . right, n n F] ** tree(n n F) ** H ** #(n n n F), tree . left, n F)) .

  eq derive((tree(L) ** H ** #(F)), L + 2)
   = (Assert(~(L === null))
      ~> answer([L, tree . data, F] ** [L + 1, tree . left, n F] ** tree(n F) ** tree(n n F) ** H ** #(n n n F), tree . right, n n F)) .

 crl (Cond,Subst) |- H' := H ** [L, tree . data, V] ** [L1, tree . left, L1'] ** tree(L1'') ** [L2, tree . right, L2'] ** tree(L2'')
  => (Cond,Subst) |- H' := H ** tree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L1' === L1'' /\ L2' === L2'') .

 crl (Cond,Subst) |- H' := H ** [L, tree . data, V] ** [L1, tree . left, L1'] ** [L2, tree . right, L2'] ** tree(L2'')
  => (Cond,Subst) |- H' := H ** tree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L1' === null /\ L2' === L2'') .

 crl (Cond,Subst) |- H' := H ** [L, tree . data, V] ** [L1, tree . left, L1'] ** tree(L1'') ** [L2, tree . right, L2']
  => (Cond,Subst) |- H' := H ** tree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L1' === L1'' /\ L2' === null) .

 crl (Cond,Subst) |- H' := H ** [L, tree . data, V] ** [L1, tree . left, L1'] ** [L2, tree . right, L2']
  => (Cond,Subst) |- H' := H ** tree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L1' === null /\ L2' === null) .


------------------
--- tree(S)(L) ---
------------------
  eq assumeCond(tree(S)(L)) = cases(Assume(S === emptyTree /\ (L === null)), Assume(~(S === emptyTree) /\ ~(L === null))) .
 ceq (Cond,Subst) |- H' := H ** tree(S)(L) = (Cond /\ S === emptyTree, Subst) |- H' := H   if VALID(Cond => (L === null)) .
 ceq (Cond,Subst) |- H' ** tree(S)(L) := H
   = (Cond,(Subst,checkAndMakeSubst(Cond,Subst,S,emptyTree))) |- H' := H   if VALID(Cond => (L === null)) .

  eq derive((tree(S)(L) ** H ** #(F)), L)
   = (Assert(~(L === null)) ~> Assume(~(S === emptyTree) /\ Data(S) === F)
      ~> answer([L + 1, tree . left, n F] ** tree(Left(S))(n F) ** [L + 2, tree . right, n n F]
                                         ** tree(Right(S))(n n F) ** H ** #(n n n F), tree . data, F)) .

  eq derive((tree(S)(L) ** H ** #(F)), L + 1)
   = (Assert(~(L === null)) ~> Assume(~(S === emptyTree) /\ Data(S) === F)
      ~> answer([L, tree . data, F] ** tree(Left(S))(n F) ** [L + 2, tree . right, n n F]
                                   ** tree(Right(S))(n n F) ** H ** #(n n n F), tree . left, n F)) .

  eq derive((tree(S)(L) ** H ** #(F)), L + 2)
   = (Assert(~(L === null))  ~> Assume(~(S === emptyTree) /\ Data(S) === F)
      ~> answer([L, tree . data, F] ** [L + 1, tree . left, n F] ** tree(Left(S))(n F)
                                                            ** tree(Right(S))(n n F) ** H ** #(n n n F), tree . right, n n F)) .

 crl (Cond,Subst) |- H' := H ** [L, tree . data, V] ** [L1, tree . left, L1'] ** tree(S1)(L1'') ** [L2, tree . right, L2'] ** tree(S2)(L2'')
  => (Cond,Subst) |- H' := H ** tree(Tree(V,S1,S2))(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L1' === L1'' /\ L2' === L2'') .

 crl (Cond,Subst) |- H' := H ** [L, tree . data, V] ** [L1, tree . left, L1'] ** [L2, tree . right, L2'] ** tree(S2)(L2'')
  => (Cond,Subst) |- H' := H ** tree(Tree(V,emptyTree,S2))(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L1' === null /\ L2' === L2'') .

 crl (Cond,Subst) |- H' := H ** [L, tree . data, V] ** [L1, tree . left, L1'] ** tree(S1)(L1'') ** [L2, tree . right, L2']
  => (Cond,Subst) |- H' := H ** tree(Tree(V,S1,emptyTree))(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L1' === L1'' /\ L2' === null) .

 crl (Cond,Subst) |- H' := H ** [L, tree . data, V] ** [L1, tree . left, L1'] ** [L2, tree . right, L2']
  => (Cond,Subst) |- H' := H ** tree(Tree(V,emptyTree,emptyTree))(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L1' === null /\ L2' === null) .



***(
--------------------------
--- generic data stack ---
--------------------------
  eq stack[HDef](L) ** #(F) = stack[HDef](L)[stack[HDef] . data = F, stack[HDef] . next = n(F)] ** #(n(n(F))) .
 ceq Cond |- H' := H ** stack[HDef](L)[?:[Fields]] = Cond |- H' := H   if VALID(Cond => (L === null)) .
 ceq Cond |- H' ** stack[HDef](L) := H = Cond |- H' := H   if VALID(Cond => (L === null)) .

  eq derive((stack[HDef](L)[stack[HDef] . data = F1, stack[HDef] . next = F2] ** H), L)
   = (Assert(~(L === null)) ~> Assume(~(F1 === null))
     ~> answer([L + 1, stack[HDef] . next, F2] ** stack[HDef](F2) ** HDef(F1) ** H, stack[HDef] . data, F1)) .

  eq derive((stack[HDef](L)[stack[HDef] . data = F1, stack[HDef] . next = F2] ** H), L + 1)
   = (Assert(~(L === null)) ~> Assume(~(F1 === null))
     ~> answer([L, stack[HDef] . data, F1] ** stack[HDef](F2) ** HDef(F1) ** H, stack[HDef] . next, F2)) .

 crl Cond |- H' := H ** [L, stack[HDef] . data, V] ** [L1, stack[HDef] . next, L2] ** stack[HDef](L')[?:[Fields]] ** HDef(V)[?':[Fields]]
  => Cond |- H' := H ** stack[HDef](L)[?]   if VALID(Cond => ~(V === null) /\ L1 === L + 1 /\ L' === L2) .

 crl Cond |- H' := H ** [L, stack[HDef] . data, V] ** [L1, stack[HDef] . next, L'] ** HDef(V)[?':[Fields]]
  => Cond |- H' := H ** stack[HDef](L)[?]   if VALID(Cond => ~(V === null) /\ L1 === L + 1 /\ L' === null) .

***)

---------------------
--- List examples ---
---------------------



  eq wrong =
--- wrong(list * a)
--- accesses * a without checking whether a is null or not
    //@ assume([list(a)]) ;
--- comment line below and it will not work
    if (a != null) {
       *(a + 1) = a ;
    }
.

---endm
---rew [| wrong |] .
---q



  eq addHead =
--- list * addHead(list * a, node * n)
--- prepends n as new head to a; n must be different from null
*** requires n != null && [(node(n) ** list(a))]
*** ensures  [list(n)]
    //@ assume n != null && [(node(n) ** list(a))] ;
    *(n + 1) = a ;
--- return(n)
    //@ assert [list(n)]
.

---endm
---rew [| addHead |] .
---q


op addHeadComplete : -> K .
  eq addHeadComplete =
--- list * addHead(list * a, node * n)
--- prepends n as new head to a; n must be different from null
*** requires [(node(e)(n) ** list(sa)(a))]
*** ensures  [list(oneElemSeq(e) :: sa)(n)]
    //@ assume [(node(e)(n) ** list(sa)(a))] ;
    *(n + 1) = a ;
--- return(n)
    //@ assert [list(oneElemSeq(e) :: sa)(n)]
.

---endm
---rew [| addHeadComplete |] .
---q



  eq allocAndAddHead =
--- list * allocAndAddHead(list * a, data k)
--- allocs a new node containing k, then links it to a and returns the larger list
*** requires [list(a)]
*** ensures  result != null && [list(result)]
    //@ assume [list(a)] ;
    x = alloc(node . data, node . next) ;
    * x = k ;
    *(x + 1) = a ;
    result = x ;
    //@ assert result != null && [list(result)]
.

---endm
---rew [| allocAndAddHead |] .
---q



op allocAndAddHeadComplete : -> K .
  eq allocAndAddHeadComplete =
--- list * allocAndAddHead(list * a, data k)
--- allocs a new node containing k, then links it to a and returns the larger list
*** requires [list(sa)(a)]
*** ensures  result != null && [list(oneElemSeq(k) :: sa)(result)]
    //@ assume [list(sa)(a)] ;
    x = alloc(node . data, node . next) ;
    * x = k ;
    *(x + 1) = a ;
    result = x ;
    //@ assert result != null && [list(oneElemSeq(k) :: sa)(result)]
.

---endm
---rew [| allocAndAddHeadComplete |] .
---q



  eq appendRec1 =
--- append1(list * a, list * b)
--- appends list b to the end of list a; b can be null, but a cannot be null
*** requires a != null && [list(a) ** list(b)]
*** ensures [list(a)]
    //@ assume a != null && [list(a) ** list(b) ** rest] ;
    if (*(a + 1) == null) {
      *(a + 1) = b ;
---   return()
      //@ assert [list(a) ** rest] ;
      stop
    } ;
```

```
--- append(*(a + 1), b)
    //@ assert *(a + 1) != null && [list(*(a + 1)) ** list(b) ** ?frame] ;
    //@ assume [list(*(a + 1)) ** ?frame] ;
--- return()
    //@ assert [list(a) ** rest]
.

---endm
---rew [| appendRec1 |] .
---q



op appendRec1Complete : -> K .
  eq appendRec1Complete =
--- append1(list * a, list * b)
--- appends list b to the end of list a; b can be null, but a cannot be null
*** requires a != null && [list(sa)(a) ** list(sb)(b)]
*** ensures [list(sa :: sb)(a)]
    //@ assume a != null && [list(sa)(a) ** list(sb)(b) ** frame] ;
    if (*(a + 1) == null) {
      *(a + 1) = b ;
---   return()
      //@ assert [list(sa :: sb)(a) ** frame] ;
      stop
    } ;
--- append(*(a + 1), b)
    //@ assert *(a + 1) != null && [list(?sa)(*(a + 1)) ** list(?sb)(b) ** ?frame] ;
    //@ assume [list(?sa :: ?sb)(*(a + 1)) ** ?frame] ;
--- return()
    //@ assert [list(sa :: sb)(a) ** frame]
.

---endm
---rew [| appendRec1Complete |] .
---q

  eq appendWhile1 =
--- append1(list * a, list * b)
--- appends list b to the end of list a; b can be null, but a cannot be null
*** requires a != null && [list(a) ** list(b)]
*** ensures [list(a)]
    //@ assume a != null && [list(a) ** list(b)] ;
    x = a ;
    //@ inv a != null && [seq(?z)(a,x) ** list(?z) ** ?frame]
    while (*(x + 1) != null) {
      x = *(x + 1) ;
    } ;
    *(x + 1) = b ;
    //@ assert [list(a)]
.

---endm
---rew [| appendWhile1 |] .
---q



ops ?sax ?sz : -> FreshVar .

op appendWhile1Complete : -> K .
  eq appendWhile1Complete =
--- append1(list * a, list * b)
--- appends list b to the end of list a; b can be null, but a cannot be null
*** requires a != null && [list(sa)(a) ** list(sb)(b)]
*** ensures [list(sa :: sb)(a)]
    //@ assume a != null && [list(sa)(a) ** list(sb)(b)] ;
    x = a ;
    //@ inv a != null && [seq(?sax,?z)(a,x) ** list(?sz)(?z) ** ?frame] && sa == ?sax :: ?sz
    while (*(x + 1) != null) {
      x = *(x + 1) ;
    } ;
    *(x + 1) = b ;
    //@ assert [list(sa :: sb)(a)]
.

---endm
---rew [| appendWhile1Complete |] .
---q
```

```
  eq appendRec2 =
--- list * append2(list * a, list * b)
--- this one returns a poiter to a list appendig a and b, so a can also be null
*** requires [list(a) ** list(b)]
*** ensures  [list(result)]
    //@ assume [list(a) ** list(b)] ;
    if (a == null) {
---   return(b) ;
      result = b ;
      //@ assert [list(result)] ;
      stop
    } ;
--- x = append(a -> next,b)
    //@ assert [list(*(a + 1)) ** list(b) ** ?frame] ;
    //@ assume [list(x) ** ?frame] ;
    *(a + 1) = x ;
--- return(a)
    result = a ;
    //@ assert [list(result)] ;
.

---endm
---rew [| appendRec2 |] .
---q


op appendRec2Complete : -> K .
eq  appendRec2Complete =
--- list * append2(list * a, list * b)
--- this one returns a poiter to a list appendig a and b, so a can also be null
*** requires [list(sa)(a) ** list(sb)(b)]
*** ensures  [list(sa :: sb)(result)]
    //@ assume [list(sa)(a) ** list(sb)(b)] ;
    if (a == null) {
---   return(b) ;
      result = b ;
      //@ assert [list(sa :: sb)(result)] ;
      stop
    } ;
--- x = append(a -> next,b)
    //@ assert [list(?sa)(*(a + 1)) ** list(?sb)(b) ** ?frame] ;
    //@ assume [list(?sa :: ?sb)(x) ** ?frame] ;
    *(a + 1) = x ;
--- return a
    result = a ;
    //@ assert [list(sa :: sb)(result)] ;
.

---endm
---rew [| appendRec2Complete |] .
---q



  eq appendWhile2 =
--- list * append2(list * a, list * b)
--- this one returns a poiter to a list appendig a and b, so a can also be null
*** requires [list(sa)(a) ** list(sb)(b)]
*** ensures  [list(sa :: sb)(result)]
    //@ assume [list(a) ** list(b)] ;
    if (a == null) {
---   return(b) ;
      result = b ;
      //@ assert [list(result)] ;
      stop
    } ;
    x = a ;
    //@ inv a != null && [seq(?z)(a,x) ** list(?z) ** ?frame]
    while (*(x + 1) != null) {
      x = *(x + 1)
    } ;
    *(x + 1) = b ;
--- return(a)
    result = a ;
    //@ assert [list(result)]
.
```

```
---endm
---rew [| appendWhile2 |] .
---q


op appendWhile2Complete : -> K .
  eq appendWhile2Complete =
--- list * append2(list * a, list * b)
--- this one returns a poiter to a list appendig a and b, so a can also be null
*** requires [list(sa)(a) ** list(sb)(b)]
*** ensures  [list(sa :: sb)(result)]
    //@ assume [list(sa)(a) ** list(sb)(b)] ;
    if (a == null) {
---   return(b) ;
      result = b ;
      //@ assert [list(sa :: sb)(result)] ;
      stop
    } ;
    x = a ;
    //@ inv a != null && [seq(?sax,?z)(a,x) ** list(?sz)(?z) ** ?frame] && sa == ?sax :: ?sz
    while (*(x + 1) != null) {
      x = *(x + 1)
    } ;
    *(x + 1) = b ;
--- return(a)
    result = a ;
    //@ assert [list(sa :: sb)(result)]
.

---endm
---rew [| appendWhile2Complete |] .
---q


  eq reverse1 =
--- list * reverse(list * a)
--- this is a "functional" implementation of return, very inefficient; uses append, though.
*** requires [list(a)]
*** ensures  [list(result)]
    //@ assume [list(a)] ;
    if (a == null) {
---   return(null)
      result = null ;
      //@ assert [list(result)] ;
      stop
    } ;
    if (*(a + 1) == null) {
---   return(a)
      result = a ;
      //@ assert [list(result)] ;
      stop
    } ;
    x = *(a + 1) ;
    *(a + 1) = null ;
--- reverse1(x)
    //@ assert [list(x) ** ?frame] ;
    //@ assume [list(x) ** ?frame] ;
--- append1(x,a) ;
    //@ assert x != null && [list(x) ** list(a) ** ?frame1] ;
    //@ assume [list(x) ** ?frame1] ;
--- return(x)
    result = x ;
    //@ assert [list(result)]
.

---endm
---rew [| reverse1 |] .
---q


ops ?sx ?sx1 : -> FreshVar .

op reverse1Complete : -> K .
  eq reverse1Complete =
--- list * reverse(list * a)
--- this is a "functional" implementation of return, very inefficient; uses append, though.
*** requires [list(sa)(a)]
*** ensures  [list(reverse(sa))(result)]
```

```
    //@ assume [list(sa)(a)] ;
    if (a == null) {
---   return(null)
      result = null ;
      //@ assert [list(reverse(sa))(result)] ;
      stop
    } ;
    if (*(a + 1) == null) {
---   return(a)
      result = a ;
      //@ assert [list(reverse(sa))(result)] ;
      stop
    } ;
    x = *(a + 1) ;
    *(a + 1) = null ;
--- reverse1(x)
    //@ assert [list(?sx)(x) ** ?frame] ;
    //@ assume [list(reverse(?sx))(x) ** ?frame] ;
--- append1(x,a) ;
    //@ assert x != null && [list(?sx1)(x) ** list(?sa)(a) ** ?frame1] ;
    //@ assume [list(?sx1 :: ?sa)(x) ** ?frame1] ;
--- return(x)
    result = x ;
    //@ assert [list(reverse(sa))(result)]
.

---endm
---rew [| reverse1Complete |] .
---q



  eq reverse2 =
--- list * reverse(list * a, list * r)
--- this is a more efficient recursive implementation of reverse, based on an auxilliary accumulator
*** requires [list(a) ** list(r)]
*** ensures  [list(result)]
    //@ assume [list(a) ** list(r)] ;
    if (a == null) {
--- return(r)
      result = r ;
      //@ assert [list(result)] ;
      stop
    } ;
    t = *(a + 1) ;
    *(a + 1) = r ;
--- return(reverse(t,a)), which is equivalent to result = reverse(t,a)
    //@ assert [list(t) ** list(a) ** ?frame] ;
    //@ assume [list(result) ** ?frame] ;
    //@ assert [list(result)]
.

---endm
---rew [| reverse2 |] .
---q

op sr : -> Var .
op ?st : -> FreshVar .

op reverse2Complete : -> K .
  eq reverse2Complete =
--- list * reverse(list * a, list * r)
--- this is a more efficient recursive implementation of reverse, based on an auxilliary accumulator
*** requires [list(sa)(a) ** list(sr)(r)]
*** ensures  [list(reverse(sa) :: sr)(result)]
    //@ assume [list(sa)(a) ** list(sr)(r)] ;
    if (a == null) {
--- return(r)
      result = r ;
      //@ assert [list(reverse(sa) :: sr)(result)] ;
      stop
    } ;
    t = *(a + 1) ;
    *(a + 1) = r ;
--- return(reverse(t,a)), which is equivalent to result = reverse(t,a)
    //@ assert [list(?st)(t) ** list(?sa)(a) ** ?frame] ;
    //@ assume [list(reverse(?st) :: ?sa)(result) ** ?frame] ;
    //@ assert [list(reverse(sa) :: sr)(result)]
.
```

```
---endm
---rew [| reverse2Complete |] .
---q



  eq reverseWhile =
--- list * reverse(list * a)
*** requires [list(a)]
*** ensures  [list(result)]
    //@ assume [list(a)] ;
    if (a == null) {
--- return(null)
      result = null ;
      //@ assert [list(result)] ;
      stop
    } ;
    x = a ;
    y = *(a + 1) ;
    *(x + 1) = null ;
    //@ inv [list(x) ** list(y) ** ?frame]
    while (y != null) {
      t = *(y + 1) ;
      *(y + 1) = x ;
      x = y ;
      y = t
    } ;
--- return(x)
    result = x ;
    //@ assert [list(result)]
.

---endm
---rew [| reverseWhile |] .
---q



op ?sy : -> FreshVar .

op reverseWhileComplete : -> K .
  eq reverseWhileComplete =
--- list * reverse(list * a)
*** requires [list(sa)(a) ** rest]
*** ensures  [list(reverse(sa))(result) ** rest]
    //@ assume [list(sa)(a) ** rest] ;
    if (a == null) {
--- return(null)
      result = null ;
      //@ assert [list(reverse(sa))(result) ** rest] ;
      stop
    } ;
    x = a ;
    y = *(a + 1) ;
    *(x + 1) = null ;
    //@ inv [list(?sx)(x) ** list(?sy)(y) ** ?frame] && reverse(sa) == reverse(?sy) :: ?sx
    while (y != null) {
      t = *(y + 1) ;
      *(y + 1) = x ;
      x = y ;
      y = t
    } ;
--- return(x)
    result = x ;
    //@ assert [list(reverse(sa))(result) ** rest]
.

---endm
---rew [| reverseWhileComplete |] .
---q



  eq disposeList =
--- disposeList(list * a)
*** requires [list(a)]
*** ensures  [empty]
    //@ assume [list(a)] ;
    if (a != null) {
---   dispose(*(a + 1))
      //@ assert [list(*(a + 1)) ** ?frame] ;
      //@ assume [?frame] ;
      free(a) ; free(a + 1) ;
    } ;
    //@ assert [empty]
.

---endm
---rew [| disposeList |] .
---q



  eq disposeListWhile =
--- disposeList(list * a)
*** requires [list(a)]
*** ensures  [empty]
    //@ assume [list(a)] ;
    //@ inv [list(a) ** ?frame]
    while (a != null) {
      b = *(a + 1) ;
      free(a) ; free(a + 1) ;
      a = b
    } ;
    //@ assert [empty]
.

---endm
---rew [| disposeListWhile |] .
---q



----------------------
--- Queue examples ---
----------------------



  eq enqueue =
--- enqueue(queue * q, node * n)
--- //@ assumes q != null and n != null, and appends n to the queue
--- if the queue is empty, i.e., * q = null and *(q+1) = null, then it initializes it with the node n
*** requires q != null && n != null && [queue(q) ** node(n)]
*** ensures  [queue(q)]
    //@ assume q != null && n != null && [queue(q) ** node(n)] ;
    if (* q == null) {
      * q = n ;
      *(q + 1) = n ;
---   return()
      //@ assert [queue(q)] ;
      stop
    } ;
    *(*(q + 1) + 1) = n ;
    *(q + 1) = n ;
--- return()
    //@ assert [queue(q)]
.

---endm
---rew [| enqueue |] .
---q



op enqueueComplete : -> K .
  eq enqueueComplete =
--- enqueue(queue * q, node * n)
--- //@ assumes q != null and n != null, and appends n to the queue
--- if the queue is empty, i.e., * q = null and *(q+1) = null, then it initializes it with the node n
*** requires q != null && n != null && [queue(sq)(q) ** node(e)(n)]
*** ensures  [queue(sq :: oneElemSeq(e))(q)]
    //@ assume q != null && n != null && [queue(sq)(q) ** node(e)(n)] ;
    if (* q == null) {
      * q = n ;
      *(q + 1) = n ;
---   return()
```

```
      //@ assert [queue(sq :: oneElemSeq(e))(q)] ;
      stop
    } ;
    *(*(q + 1) + 1) = n ;
    *(q + 1) = n ;
--- return()
    //@ assert [queue(sq :: oneElemSeq(e))(q)]
.

---endm
---rew [| enqueueComplete |] .
---q



  eq dequeue =
--- node * dequeue(queue * q)
--- if q == null or q empty it returns null
--- if q has elements then it returns the first element and modifies q into "q - n"
*** requires [queue(q)]
*** ensures  [queue(q) ** node(return)]
    //@ assume [queue(q)] ;
    if (q == null) {
---   return(null)
      return = null ;
      //@ assert [queue(q) ** node(return)] ;
      stop
    } ;
    if (* q == null) {
---   return(null)
      return = null ;
      //@ assert [queue(q) ** node(return)] ;
      stop
    } ;
    if (* q == *(q + 1)) {
      r = * q ;
      * q = null ;
      *(q + 1) = null ;
--- a bit of help here: access(exp) does nothing but symbolically evaluates exp and then discards its value.
--- the reason for doing so is to "roll" the heap; this is needed for the next //@ assert, which needs r to be
--- extracted from the sequence associated to the queue in order to unroll it into a node.
--- I could also do this automatically whenever anything else fails, but for now this is acceptable.
      //@ access(* r) ;  --- the followig also works:           * r = * r ;
---   return(r)
      return = r ;
      //@ assert [queue(q) ** node(return)] ;
      stop
    } ;
    r = * q ;
    * q = *(r + 1) ;
--- return(r)
    //@ assert [queue(q) ** node(r)]
.

---endm
---rew [| dequeue |] .
---q



op dequeueComplete : -> K .
  eq dequeueComplete =
--- node * dequeue(queue * q)
--- if q == null or q empty it returns null
--- if q has elements then it returns the first element and modifies q into "q - n"
*** requires [queue(sq)(q)]
*** ensures  sq == nil && result == null && [queue(nil)(q)] || sq != nil && [queue(tail(sq))(q) ** node(head(sq))(result)]
    //@ assume [queue(sq)(q)] ;
    if (q == null) {
---   return(null)
      result = null ;
      //@ assert sq == nil && result == null && [queue(nil)(q)] || sq != nil && [queue(tail(sq))(q) ** node(head(sq))(result)] ;
      stop
    } ;
    if (* q == null) {
---   return(null)
      result = null ;
      //@ assert sq == nil && result == null && [queue(nil)(q)] || sq != nil && [queue(tail(sq))(q) ** node(head(sq))(result)] ;
      stop
    } ;
    if (* q == *(q + 1)) {
      r = * q ;
      * q = null ;
      *(q + 1) = null ;
--- a bit of help here: access(exp) does nothing but symbolically evaluates exp and then discards its value.
--- the reason for doing so is to "roll" the heap; this is needed for the next //@ assert, which needs r to be
--- extracted from the sequence associated to the queue in order to unroll it into a node.
--- I could also do this automatically whenever anything else fails, but for now this is acceptable.
      //@ access(* r) ;  --- the followig also works:           * r = * r ;
---   return(r)
      result = r ;
      //@ assert sq == nil && result == null && [queue(nil)(q)] || sq != nil && [queue(tail(sq))(q) ** node(head(sq))(result)] ;
      stop
    } ;
    r = * q ;
    * q = *(r + 1) ;
--- return(r)
    result = r ;
    //@ assert sq == nil && result == null && [queue(nil)(q)] || sq != nil && [queue(tail(sq))(q) ** node(head(sq))(result)] ;
.

---endm
---rew [| dequeueComplete |] .
---q



  eq transferOwner1 =
--- transferOwner1(queue * from, queue * to)
--- dequeues "from" and enqueues the resulting node to "to"
--- we prove that it all takes the same heap space
--- everything is done manually here, without "calling" enqueue and dequeue defined above
*** requires [queue(from) ** queue(to)]
*** ensures  [queue(from) ** queue(to)]
    //@ assume [queue(from) ** queue(to)] ;
    if (from == null || to == null) {
---   return()
      //@ assert [queue(from) ** queue(to)] ;
      stop ;
    } ;
    if (* from == null) {
---   return()
      //@ assert [queue(from) ** queue(to)] ;
      stop ;
    } ;
    n = * from ;
    if (*(from + 1) == n) {
      * from = null ;
      *(from + 1) = null ;
    } else {
      * from = *(n + 1)
    } ;
    if (* to == null) {
      * to = n
    } else {
      *(*(to + 1) + 1) = n
    } ;
    *(to + 1) = n ;
--- again, a bit of help to unroll the location n in the heap
    //@ access(* n) ;
--- return()
    //@ assert [queue(from) ** queue(to)]
.

---endm
---rew [| transferOwner1 |] .
---q



op transferOwner1Complete : -> K .
  eq transferOwner1Complete =
--- transferOwner1(queue * from, queue * to)
--- dequeues "from" and enqueues the resulting node to "to"
--- we prove that it all takes the same heap space
--- everything is done manually here, without "calling" enqueue and dequeue defined above
*** requires [queue(sf)(from) ** queue(st)(to)]
*** ensures (sf == nil || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(tail(sf))(from) ** queue(st :: oneElemSeq(head(sf)))(to)]
```

```
    //@ assume [queue(sf)(from) ** queue(st)(to)] ;
    if (from == null || to == null) {
---   return()
      //@ assert (sf == nil || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(tail(sf))(from) ** queue(st :: oneElemSeq(head(sf)))(to)] ;
      stop ;
    } ;
    if (* from == null) {
---   return()
      //@ assert (sf == nil || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(tail(sf))(from) ** queue(st :: oneElemSeq(head(sf)))(to)] ;
      stop ;
    } ;
    n = * from ;
    if (*(from + 1) == n) {
      * from = null ;
      *(from + 1) = null ;
    } else {
      * from = *(n + 1)
    } ;
    if (* to == null) {
      * to = n
    } else {
      *(*(to + 1) + 1) = n
    } ;
    *(to + 1) = n ;
--- again, a bit of help to unroll the location n in the heap
    //@ access(* n) ;
--- return()
    //@ assert (sf == nil || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(tail(sf))(from) ** queue(st :: oneElemSeq(head(sf)))(to)]
.

---endm
---rew [| transferOwner1Complete |] .
---q



eq transferOwner2 =
--- transferOwner2(queue * from, queue * to)
--- same as above, but it "calls" dequeue and enqueue
*** requires [queue(from) ** queue(to)]
*** ensures  [queue(from) ** queue(to)]
    //@ assume [queue(from) ** queue(to)] ;
    if (from == null || to == null) {
---   return()
      //@ assert [queue(from) ** queue(to)] ;
      stop ;
    } ;
---    n = dequeue(from) ;
    //@ assert [queue(from) ** ?rest1] ;
    //@ assume [queue(from) ** node(n) ** ?rest1] ;
    if (n == null) {
---   return()
      //@ assert [queue(from) ** queue(to)] ;
      stop ;
    } ;
---    enqueue(to,n) ;
    //@ assert to != null && n != null && [queue(to) ** node(n) ** ?rest2] ;
    //@ assume [queue(to) ** ?rest2] ;
---    return()
    //@ assert [queue(from) ** queue(to)]
.

---endm
---rew [| transferOwner2 |] .
---q




op transferOwner2Complete : -> K .
eq transferOwner2Complete =
--- transferOwner2(queue * from, queue * to)
--- same as above, but it "calls" dequeue and enqueue
*** requires [queue(sf)(from) ** queue(st)(to)]
*** ensures (sf == nil || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(tail(sf))(from) ** queue(st :: oneElemSeq(head(sf)))(to)]
    //@ assume [queue(sf)(from) ** queue(st)(to)] ;
    if (from == null || to == null) {
---   return()
```

```
      //@ assert (sf == nil || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(tail(sf))(from) ** queue(st :: oneElemSeq(head(sf)))(to)] ;
      stop ;
    } ;
---    n = dequeue(from) ;
    //@ assert [queue(?sf)(from) ** ?rest1] ;
    //@ assume ?sf == nil && n == null && [queue(nil)(from) ** ?rest1] || ?sf != nil && [queue(tail(?sf))(from) ** node(head(?sf))(n) ** ?rest1] ;
    if (n == null) {
---   return()
      //@ assert (sf == nil || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(tail(sf))(from) ** queue(st :: oneElemSeq(head(sf)))(to)] ;
      stop ;
    } ;
---    enqueue(to,n) ;
    //@ assert to != null && n != null && [queue(?st)(to) ** node(?e)(n) ** ?rest2] ;
    //@ assume [queue(?st :: oneElemSeq(?e))(to) ** ?rest2] ;
---    return()
    //@ assert (sf == nil || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(tail(sf))(from) ** queue(st :: oneElemSeq(head(sf)))(to)]
.

---endm
---rew [| transferOwner2Complete |] .
---q



  eq stealQueue =
--- stealQueue(queue * from, queue * to)
--- steals queue "from" by transfering one element from it to "to"
--- it speculates the fact that a dequed element still keeps the pointer to the rest of the queue
--- thus, after transfering the dequeued element from "from" to "to", the two queues are sequentialized
--- all we have to do is to move the tail of "to" to the tail of "from" after the transfer
--- ... apparently ... one also has to free the two pointers held by "from" ... in case "from" was not empty
*** requires [queue(from) ** queue(to)]
*** ensures  (from == null || to == null) && [queue(from) ** queue(to)] || [queue(to)]
    //@ assume [queue(from) ** queue(to)] ;
    if (from == null || to == null) {
---   return()
      //@ assert (from == null || to == null) && [queue(from) ** queue(to)] || [queue(to)] ;
      stop ;
    } ;
    if (* from != null) {
      if (* to == null) {
        * to = * from
      } else {
        *(*(to + 1) + 1) = * from ;
      } ;
      *(to + 1) = *(from + 1) ;
    } ;
    free(from) ; free(from + 1) ;
    //@ assert (from == null || to == null) && [queue(from) ** queue(to)] || [queue(to)] ;
.

---endm
---rew [| stealQueue |] .
---q



op stealQueueComplete : -> K .
  eq stealQueueComplete =
--- stealQueue(queue * from, queue * to)
--- steals queue "from" by transfering one element from it to "to"
--- it speculates the fact that a dequed element still keeps the pointer to the rest of the queue
--- thus, after transfering the dequeued element from "from" to "to", the two queues are sequentialized
--- all we have to do is to move the tail of "to" to the tail of "from" after the transfer
--- ... apparently ... one also has to free the two pointers held by "from" ... in case "from" was not empty
*** requires [queue(sf)(from) ** queue(st)(to)]
*** ensures  (from == null || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(st :: sf)(to)]
    //@ assume [queue(sf)(from) ** queue(st)(to)] ;
    if (from == null || to == null) {
---   return()
      //@ assert (from == null || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(st :: sf)(to)] ;
      stop ;
    } ;
    if (* from != null) {
      if (* to == null) {
        * to = * from
      } else {
        *(*(to + 1) + 1) = * from ;
      } ;
      *(to + 1) = *(from + 1) ;
    } ;
    free(from) ; free(from + 1) ;
    //@ assert (from == null || to == null) && [queue(sf)(from) ** queue(st)(to)] || [queue(st :: sf)(to)]
.

---endm
---rew [| stealQueueComplete |] .
---q



---------------------
--- Tree examples ---
---------------------



eq allocTree =
--- tree * allocTree(data k, tree * left, tree * right)
*** requires [tree(left) ** tree(right)]
*** ensures  [tree(result)]
    //@ assume [tree(left) ** tree(right)] ;
    t = alloc(tree . data, tree . left, tree . right) ;
    * t = k ;
    *(t + 1) = left ;
    *(t + 2) = right ;
--- return(t)
    result = t ;
    //@ assert [tree(result)]
.

---endm
---rew [| allocTree |] .
---q



op allocTreeComplete : -> K .
eq allocTreeComplete =
*** requires [tree(tl)(left) ** tree(tr)(right)]
*** ensures  [tree(Tree(k,tl,tr))(result)]
    //@ assume [tree(tl)(left) ** tree(tr)(right)] ;
    t = alloc(tree . data, tree . left, tree . right) ;
    * t = k ;
    *(t + 1) = left ;
    *(t + 2) = right ;
--- return(t)
    result = t ;
    //@ assert [tree(Tree(k,tl,tr))(result)]
.

---endm
---rew [| allocTreeComplete |] .
---q



eq mirror =
--- mirror(tree * t)
--- mirrors the tree t
    //@ assume [tree(t)] ;
    if (t == null) {
---   return()
      //@ assert [tree(t)] ;
      stop ;
    } ;
--- mirror(*(t + 1))
    //@ assert [tree(*(t + 1)) ** ?rest1] ;
    //@ assume [tree(*(t + 1)) ** ?rest1] ;
--- mirror(*(t + 2))
    //@ assert [tree(*(t + 2)) ** ?rest2] ;
    //@ assume [tree(*(t + 2)) ** ?rest2] ;
--- swap the left and right subtrees
*** this is wrong and it catches it!
--- *(t + 1) = *(t + 2) ;
--- *(t + 2) = *(t + 1) ;
*** this is correct, using a temporary
    x = *(t + 1) ;
    *(t + 1) = *(t + 2) ;
    *(t + 2) = x ;
--- return()
    //@ assert [tree(t)] ;
.

---endm
---rew [| mirror |] .
---q


op mirrorComplete : -> K .
eq mirrorComplete =
--- mirror(tree * t)
--- mirrors the tree t
*** requires [tree(tr)(t)]
*** ensures  [tree(mirror(tr))(t)]
    //@ assume [tree(tr)(t)] ;
    if (t == null) {
---   return()
      //@ assert [tree(t)] ;
      stop ;
    } ;
--- mirror(*(t + 1))
    //@ assert [tree(?tl)(*(t + 1)) ** ?rest1] ;
    //@ assume [tree(mirror(?tl))(*(t + 1)) ** ?rest1] ;
--- mirror(*(t + 2))
    //@ assert [tree(?tr)(*(t + 2)) ** ?rest2] ;
    //@ assume [tree(mirror(?tr))(*(t + 2)) ** ?rest2] ;
--- swap the left and right subtrees
*** this is wrong and it catches it!
--- *(t + 1) = *(t + 2) ;
--- *(t + 2) = *(t + 1) ;
*** this is correct, using a temporary
    x = *(t + 1) ;
    *(t + 1) = *(t + 2) ;
    *(t + 2) = x ;
--- return()
    //@ assert [tree(mirror(tr))(t)] ;
.

---endm
---rew [| mirrorComplete |] .
---q

***(

eq treeToList1 =
--- list * treeToList(tree * t)
--- traverses t in infix order and returns a list
--- also, it deallocates t at the same time
--- this uses append, so it is inefficient
*** requires [tree(t)]
*** ensures  [list(return)]
    //@ assume [tree(t)] ;
    if (t == null) {
---   return(null)
      return = null ;
      //@ assert [list(return)] ;
      stop ;
    } ;
--- x = treeToList(t -> left)
    //@ assert [tree(*(t + 1)) ** rest1] ;
    //@ assume [list(x) ** rest1] ;
--- y = treeToList(t -> right)
    //@ assert [tree(*(t + 2)) ** rest2] ;
    //@ assume [list(y) ** rest2] ;
    z = alloc(node . data,node . next) ;
    * z = *(t) ;
    *(z + 1) = y ;
    free(t); free(t + 1) ; free(t + 2) ;
--- r = append2(x,z)
    //@ assert [list(x) ** list(z) ** rest3] ;
    //@ assume [list(r) ** rest3] ;
--- return(r)
    return = r ;
    //@ assert [list(return)]
.
```

```
---endm
---rew [| treeToList1 |] .
---q



eq treeToList2 =
--- list * treeToList(tree * t, list * l)
--- an efficient version of the above, based on an accumulator list l
*** requires [tree(t) ** list(l)]
*** ensures  [list(return)]
    //@ assume [tree(t) ** list(l)] ;
    if (t == null) {
---   return(l)
      return = l ;
      //@ assert [list(return)] ;
      stop ;
    } ;
--- y = treeToList(t -> right, l)
    //@ assert [tree(*(t + 2)) ** list(l) ** rest1] ;
    //@ assume [list(y) ** rest1] ;
    left = *(t + 1) ;
    z = alloc(node . data,node . next) ;
    * z = * t ;
    *(z + 1) = y ;
    free(t); free(t + 1) ; free(t + 2) ;
--- r = treeToList(t -> left, z)
    //@ assert [tree(left) ** list(z) ** rest2] ;
    //@ assume [list(r) ** rest2] ;
--- return(r)
    return = r ;
    //@ assert [list(return)]
.

---endm
---rew [| treeToList2 |] .
---q



eq treeToListWhile =
--- list * treeToListWhile(tree * t)
*** requires [tree(t)]
*** ensures  [list(return)]
    //@ assume [tree(t)] ;
    if (t == null) {
---   return(null)
      return = null ;
      //@ assert [list(return)] ;
      stop ;
    } ;
    s = alloc(stack[tree] . data, stack[tree] . next) ;   --- stack
    * s = t ;
    *(s + 1) = null ;
    ret = null ;   --- result list
    //@ inv [stack[tree](s) ** list(ret) ** rest]
    while (s != null) {
***   t = pop(s)
      t = * s ;
      temp = s ;
      s = *(s + 1) ;
      free(temp) ;
      free(temp + 1) ;
--- get left and right subtrees, then cut the links to them in the current node
      l = *(t + 1) ;
      r = *(t + 2) ;
      *(t + 1) = null ;
      *(t + 2) = null ;

      if (l != null) {
        x = alloc(stack[tree] . data, stack[tree] . next) ;
        * x = l ;
        *(x + 1) = s ;
        s = x ;
      } ;

      if (r != null) {
        x = alloc(stack[tree] . data, stack[tree] . next) ;
        * x = t ;
        *(x + 1) = s ;
        y = alloc(stack[tree] . data, stack[tree] . next) ;
        * y = r ;
        *(y + 1) = x ;
        s = y ;
      } else {
        z = alloc(node . data, node . next) ;
        * z = * t ;
        free(t) ;
        free(t + 1) ;
        free(t + 2) ;
        *(z + 1) = ret ;
        ret = z ;
      }
    } ;
--- return(ret)
    return = ret ;
    //@ assert [list(return)]
.

---endm
---rew [| treeToListWhile |] .
---q

***)


***(
--------------------
--- Schorr-Waite ---
--------------------

-------------------------------------------------
--- cleanTree(L), markedTree(L), stackTree(L) ---
-------------------------------------------------

--------------------
--- cleanTree(L) ---
--------------------

  eq //@ assumeCond(cleanTree(L)) = dot .

 ceq (Cond,Subst) |- H' := H ** cleanTree(L) = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .
 ceq (Cond,Subst) |- H' ** cleanTree(L) := H = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .

  eq derive((cleanTree(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null))
     ~> answer([L + 1, tree . switch, undef] ** [L + 2, tree . left, F] ** [L + 3, tree . right, n F]
              ** cleanTree(F) ** cleanTree(n F) ** H ** #(n n F), tree . marked, 0)) .

  eq derive((cleanTree(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null))
     ~> answer([L, tree . marked, 0] ** [L + 2, tree . left, F] ** [L + 3, tree . right, n F]
              ** cleanTree(F) ** cleanTree(n F) ** H ** #(n n F), tree . switch, undef)) .

  eq derive((cleanTree(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null))
     ~> answer([L, tree . marked, 0] ** [L + 1, tree . switch, undef] ** [L + 3, tree . right, n F]
              ** cleanTree(F) ** cleanTree(n F) ** H, tree . left, F)) .

  eq derive((cleanTree(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null))
     ~> answer([L, tree . marked, 0] ** [L + 1, tree . switch, undef] ** [L + 2, tree . left, F]
              ** cleanTree(F) ** cleanTree(n F) ** H, tree . right, n F)) .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 0] ** [L1, tree . switch, undef] ** [L2, tree . left, D1] ** [L3, tree . right, D2]
                           ** cleanTree(D1') ** cleanTree(D2')
  => (Cond,Subst) |- H' := H ** cleanTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 0] ** [L1, tree . switch, undef] ** [L2, tree . left, D1] ** [L3, tree . right, D2] ** cleanTree(D1')
  => (Cond,Subst) |- H' := H ** cleanTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === null) .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 0] ** [L1, tree . switch, undef] ** [L2, tree . left, D1] ** [L3, tree . right, D2] ** cleanTree(D2')
  => (Cond,Subst) |- H' := H ** cleanTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === null /\ D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 0] ** [L1, tree . switch, undef] ** [L2, tree . left, D1] ** [L3, tree . right, D2]
  => (Cond,Subst) |- H' := H ** cleanTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === null /\ D2 === null) .
```

```
---------------------
--- markedTree(L) ---
---------------------
  eq //@ assumeCond(markedTree(L)) = dot .

 ceq (Cond,Subst) |- H' := H ** markedTree(L) = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .
 ceq (Cond,Subst) |- H' ** markedTree(L) := H = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .

  eq derive((markedTree(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null))
     ~> answer([L + 1, tree . switch, 1] ** [L + 2, tree . left, F] ** [L + 3, tree . right, n F]
              ** markedTree(F) ** markedTree(n F) ** H ** #(n n F), tree . marked, 1)) .

  eq derive((markedTree(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null))
     ~> answer([L, tree . marked, 1] ** [L + 2, tree . left, F] ** [L + 3, tree . right, n F]
              ** markedTree(F) ** markedTree(n F) ** H ** #(n n F), tree . switch, 1)) .

  eq derive((markedTree(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null))
     ~> answer([L, tree . marked, 1] ** [L + 1, tree . switch, 1] ** [L + 3, tree . right, n F]
              ** markedTree(F) ** markedTree(n F) ** H ** #(n n F), tree . left, F)) .

  eq derive((markedTree(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null))
     ~> answer([L, tree . marked, 1] ** [L + 1, tree . switch, 1] ** [L + 2, tree . left, F]
              ** markedTree(F) ** markedTree(n F) ** H ** #(n n F), tree . right, n F)) .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 1] ** [L2, tree . left, D1] ** [L3, tree . right, D2]
                             ** markedTree(D1') ** markedTree(D2')
  => (Cond,Subst) |- H' := H ** markedTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 1] ** [L2, tree . left, D1] ** [L3, tree . right, D2] ** markedTree(D1')
  => (Cond,Subst) |- H' := H ** markedTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === null) .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 1] ** [L2, tree . left, D1] ** [L3, tree . right, D2] ** markedTree(D2')
  => (Cond,Subst) |- H' := H ** markedTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === null /\ D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 1] ** [L2, tree . left, D1] ** [L3, tree . right, D2]
  => (Cond,Subst) |- H' := H ** markedTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === null /\ D2 === null) .


--------------------
--- stackTree(L) ---
--------------------
  eq //@ assumeCond(stackTree(L)) = dot .

 ceq (Cond,Subst) |- H' := H ** stackTree(L) = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .
 ceq (Cond,Subst) |- H' ** stackTree(L) := H = (Cond,Subst) |- H' := H   if VALID(Cond => (L === null)) .

  eq derive((stackTree(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null))
    ~> cases(answer([L + 1, tree . switch, 0] ** [L + 2, tree . left, F] ** [L + 3, tree . right, n F]
                  ** stackTree(F) ** cleanTree(n F) ** H ** #(n n F), tree . marked, 1),
             answer([L + 1, tree . switch, 1] ** [L + 2, tree . left, F] ** [L + 3, tree . right, n F]
                  ** markedTree(F) ** stackTree(n F) ** H ** #(n n F), tree . marked, 1))) .

  eq derive((stackTree(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null))
    ~> cases(answer([L, tree . marked, 1] ** [L + 2, tree . left, F] ** [L + 3, tree . right, n F]
                  ** stackTree(F) ** cleanTree(n F) ** H ** #(n n F), tree . switch, 0),
             answer([L, tree . marked, 1] ** [L + 2, tree . left, F] ** [L + 3, tree . right, n F]
                  ** markedTree(F) ** stackTree(n F) ** H ** #(n n F), tree . switch, 1))) .

  eq derive((stackTree(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null))
    ~> cases(answer([L, tree . marked, 1] ** [L + 1, tree . switch, 0] ** [L + 3, tree . right, n F]
                  ** stackTree(F) ** cleanTree(n F) ** H ** #(n n F), tree . left, F),
             answer([L, tree . marked, 1] ** [L + 1, tree . switch, 1] ** [L + 3, tree . right, n F]
                  ** markedTree(F) ** stackTree(n F) ** H ** #(n n F), tree . left, F))) .

  eq derive((stackTree(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null))
    ~> cases(answer([L, tree . marked, 1] ** [L + 1, tree . switch, 0] ** [L + 2, tree . left, F]
                  ** stackTree(F) ** cleanTree(n F) ** H ** #(n n F), tree . right, n F),
             answer([L, tree . marked, 1] ** [L + 1, tree . switch, 1] ** [L + 2, tree . left, F]
                  ** markedTree(F) ** stackTree(n F) ** H ** #(n n F), tree . right, n F))) .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 0] ** [L2, tree . left, D1] ** [L3, tree . right, D2]
```

```
                             ** stackTree(D1') ** cleanTree(D2')
  => (Cond,Subst) |- H' := H ** stackTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === D2') .
 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 0] ** [L2, tree . left, D1] ** [L3, tree . right, D2] ** stackTree(D1')
  => (Cond,Subst) |- H' := H ** stackTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === null) .
 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 0] ** [L2, tree . left, D1] ** [L3, tree . right, D2] ** cleanTree(D2')
  => (Cond,Subst) |- H' := H ** stackTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === null /\ D2 === D2') .
 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 0] ** [L2, tree . left, D1] ** [L3, tree . right, D2]
  => (Cond,Subst) |- H' := H ** stackTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === null /\ D2 === null) .

 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 1] ** [L2, tree . left, D1] ** [L3, tree . right, D2]
                             ** markedTree(D1') ** stackTree(D2')
  => (Cond,Subst) |- H' := H ** stackTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === D2') .
 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 1] ** [L2, tree . left, D1] ** [L3, tree . right, D2] ** markedTree(D1')
  => (Cond,Subst) |- H' := H ** stackTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === null) .
 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 1] ** [L2, tree . left, D1] ** [L3, tree . right, D2] ** stackTree(D2')
  => (Cond,Subst) |- H' := H ** stackTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === null /\ D2 === D2') .
 crl (Cond,Subst) |- H' := H ** [L, tree . marked, 1] ** [L1, tree . switch, 1] ** [L2, tree . left, D1] ** [L3, tree . right, D2]
  => (Cond,Subst) |- H' := H ** stackTree(L)   if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === null /\ D2 === null) .

***)

------------------------------
--- Schorr-Waite with tree ---
------------------------------


eq Schorr-Waite-tree =
*** requires [cleanTree(root)]
*** esures   [markedTree(root)]
    //@ assume [cleanTree(root)] ;
    t = root ;
    p = null ;
//@ inv (t == null && [stackTree(p) ** ?rest])
        || (* t == 1 &&& [markedTree(t) ** stackTree(p) ** ?rest])
        || (* t == 0 &&& [cleanTree(t) ** stackTree(p) ** ?rest])
    while (p != null || (t != null && * t == 0)) {
      if (t == null || * t == 1) {
        if (*(p + 1) == 1) {       --- POP
          q = t ;                  --- q = t
          t = p ;                  --- t = p
          p = *(p + 3) ;           --- p = p -> right
          *(t + 3) = q ;           --- t -> right = q
        }
        else {                     --- SWING
          q = t ;                  --- q = t
          t = *(p + 3) ;           --- t = p -> right
          *(p + 3) = *(p + 2) ;    --- p -> right = p -> left
          *(p + 2) = q ;           --- p -> left = q
          *(p + 1) = 1 ;           --- p -> c = 1
        }
      }
      else {                       --- PUSH
        q = p ;                    --- q = p
        p = t ;                    --- p = t
        t = *(t + 2) ;             --- t = t -> left
        *(p + 2) = q ;             --- p -> left = q
        * p = 1 ;                  --- p -> m = 1
        *(p + 1) = 0 ;             --- p -> c = 0
      }
    } ;
    //@ assert [markedTree(t)]
.

---endm
---rew [| Schorr-Waite-tree |] .
---q


***(
---------------------------------------
--- Schorr-Waite Complete with tree ---
---------------------------------------


op Schorr-Waite-tree-Complete : -> K .
eq Schorr-Waite-tree-Complete =
*** requires [cleanTree(initTree)(root)]
*** ensures  [markedTree(initTree)(root)]
    //@ assume [cleanTree(initTree)(root)] ;
    t = root ;
    p = null ;
//@ inv t == null && [stackTree(?treeContext)(p) ** ?rest] && initTree == ?treeContext[emptyTree]
        || (* t == 1 &&& [markedTree(?tree)(t) ** stackTree(?treeContext)(p) ** ?rest] ||
            * t == 0 &&& [ cleanTree(?tree)(t) ** stackTree(?treeContext)(p) ** ?rest]) && initTree == ?treeContext[?tree]
    while (p != null || (t != null && * t == 0)) {
      if (t == null || * t == 1) {
        if (*(p + 1) == 1) {       --- POP
          q = t ;                  --- q = t
          t = p ;                  --- t = p
          p = *(p + 3) ;           --- p = p -> right
          *(t + 3) = q ;           --- t -> right = q
        }
        else {                     --- SWING
          q = t ;                  --- q = t
          t = *(p + 3) ;           --- t = p -> right
          *(p + 3) = *(p + 2) ;    --- p -> right = p -> left
          *(p + 2) = q ;           --- p -> left = q
          *(p + 1) = 1 ;           --- p -> c = 1
        }
      }
      else {                       --- PUSH
        q = p ;                    --- q = p
        p = t ;                    --- p = t
        t = *(t + 2) ;             --- t = t -> left
        *(p + 2) = q ;             --- p -> left = q
        * p = 1 ;                  --- p -> m = 1
        *(p + 1) = 0 ;             --- p -> c = 0
      }
    } ;
    //@ assert [markedTree(initTree)(t)]
.

endm
rew [| Schorr-Waite-tree-Complete |] .
q
***)

***(

----------------------------------------------------
--- cleanGraph(L), markedGraph(L), stackGraph(L) ---
----------------------------------------------------

  ops cleanGraph markedGraph stackInGraph : -> HeapDefName .
  op Schorr-Waite-graph : -> K .

---------------------
--- cleanGraph(L) ---
---------------------
--- test: remove {L} from Out sets!
  eq //@ assumeCond(cleanGraph(In,Out)(L)) = cases(//@ Assume(In === emptySet), //@ Assume(~(In === emptySet) /\ ~(L === null))) .

 ceq (Cond,Subst) |- H' := H ** cleanGraph(In,Out)(L) = (Cond /\ In === emptySet, Subst) |- H' := H
    if VALID(Cond => L === null) or VALID(Cond => oneElemSet(L) IN Out) .
 ceq (Cond,Subst) |- H' ** cleanGraph(In,Out)(L) := H
   = (Cond,(Subst,checkAndMakeSubst(Cond,Subst,In,emptySet))) |- H'[checkAndMakeSubst(Cond,Subst,In,emptySet)] := H
    if VALID(Cond => L === null) or VALID(Cond => oneElemSet(L) IN Out) .

  eq derive((cleanGraph(In,Out)(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> answer([L + 1, graph . switch, undef] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
              ** cleanGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
              ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . marked, 0)) .

  eq derive((cleanGraph(In,Out)(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> answer([L, graph . marked, 0] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
              ** cleanGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
              ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . switch, undef)) .

  eq derive((cleanGraph(In,Out)(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> answer([L, graph . marked, 0] ** [L + 1, graph . switch, undef] ** [L + 3, graph . right, n F]
              ** cleanGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
              ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . left, F)) .
  eq derive((cleanGraph(In,Out)(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> answer([L, graph . marked, 0] ** [L + 1, graph . switch, undef] ** [L + 2, graph . left, F]
              ** cleanGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
              ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . right, n F)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 0] ** [L1, graph . switch, undef] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                           ** cleanGraph(In1,Out1)(D1') ** cleanGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** cleanGraph(oneElemSet(L) UNION In1 UNION In2, (Out1 UNION Out2) MINUS (In1 UNION In2 UNION oneElemSet(L)))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
       and VALID(Cond => D1 === D1') and VALID(Cond => D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 0] ** [L1, graph . switch, undef] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                           ** cleanGraph(In1,Out1)(D1')
  => (Cond,Subst) |- H' := H ** cleanGraph(oneElemSet(L) UNION In1, Out1 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D1 === D1')
       and (VALID(Cond => D2 === null) or VALID(Cond => oneElemSet(D2) IN In1) or VALID(Cond => D2 === L) or VALID(Cond => oneElemSet(D2) IN Out1)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 0] ** [L1, graph . switch, undef] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                           ** cleanGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** cleanGraph(oneElemSet(L) UNION In2, Out2 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D2 === D2')
       and (VALID(Cond => D1 === null) or VALID(Cond => oneElemSet(D1) IN In2) or VALID(Cond => D1 === L) or VALID(Cond => oneElemSet(D1) IN Out2)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 0] ** [L1, graph . switch, undef] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
  => (Cond,Subst) |- H' := H ** cleanGraph(oneElemSet(L), emptySet)(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
       and (VALID(Cond => D1 === null) or VALID(Cond => D1 === L)) and (VALID(Cond => D2 === null) or VALID(Cond => D2 === L)) .



----------------------
--- markedGraph(L) ---
----------------------
  eq //@ assumeCond(markedGraph(In,Out)(L)) = cases(//@ Assume(In === emptySet), //@ Assume(~(In === emptySet) /\ ~(L === null))) .

 ceq (Cond,Subst) |- H' := H ** markedGraph(In,Out)(L) = (Cond /\ In === emptySet, Subst) |- H' := H
    if  VALID(Cond => L === null) or VALID(Cond => oneElemSet(L) IN Out) .
 ceq (Cond,Subst) |- H' ** markedGraph(In,Out)(L) := H
   = (Cond,(Subst,checkAndMakeSubst(Cond,Subst,In,emptySet))) |- H'[checkAndMakeSubst(Cond,Subst,In,emptySet)] := H
    if  VALID(Cond => L === null) or VALID(Cond => oneElemSet(L) IN Out) .

  eq derive((markedGraph(In,Out)(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In) ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F))
      ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> answer([L + 1, graph . switch, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
              ** markedGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
              ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . marked, 1)) .

  eq derive((markedGraph(In,Out)(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In) ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F))
      ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> answer([L, graph . marked, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
              ** markedGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
              ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . switch, 1)) .

  eq derive((markedGraph(In,Out)(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> answer([L, graph . marked, 1] ** [L + 1, graph . switch, 1] ** [L + 3, graph . right, n F]
              ** markedGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
              ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . left, F)) .

  eq derive((markedGraph(In,Out)(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> answer([L, graph . marked, 1] ** [L + 1, graph . switch, 1] ** [L + 2, graph . left, F]
              ** markedGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
              ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . right, n F)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                           ** markedGraph(In1,Out1)(D1') ** markedGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** markedGraph(oneElemSet(L) UNION In1 UNION In2, (Out1 UNION Out2) MINUS (In1 UNION In2 UNION oneElemSet(L)))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
       and VALID(Cond => D1 === D1') and VALID(Cond => D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
```

```
                             ** markedGraph(In1,Out1)(D1')
  => (Cond,Subst) |- H' := H ** markedGraph(oneElemSet(L) UNION In1, Out1 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D1 === D1')
       and (VALID(Cond => D2 === null) or VALID(Cond => oneElemSet(D2) IN In1) or VALID(Cond => D2 === L) or VALID(Cond => oneElemSet(D2) IN Out1)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                             ** markedGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** markedGraph(oneElemSet(L) UNION In2, Out2 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D2 === D2')
       and (VALID(Cond => D1 === null) or VALID(Cond => oneElemSet(D1) IN In2) or VALID(Cond => D1 === L) or VALID(Cond => oneElemSet(D1) IN Out2)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
  => (Cond,Subst) |- H' := H ** markedGraph(oneElemSet(L), emptySet)(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
       and (VALID(Cond => D1 === null) or VALID(Cond => D1 === L)) and (VALID(Cond => D2 === null) or VALID(Cond => D2 === L)) .



-----------------------
--- stackInGraph(L) ---
-----------------------

  eq //@ assumeCond(stackInGraph(In,Out)(L)) = cases(//@ Assume(In === emptySet), //@ Assume(~(In === emptySet) /\ ~(L === null))) .

 ceq (Cond,Subst) |- H' := H ** stackInGraph(In,Out)(L) = (Cond /\ In === emptySet /\ Out === emptySet, Subst) |- H' := H
    if VALID(Cond => L === null) .
 ceq (Cond,Subst) |- H' ** stackInGraph(In,Out)(L) := H
   = (Cond,(Subst,checkAndMakeSubst(Cond,Subst,(In,Out),(emptySet,emptySet)))) |- H'[checkAndMakeSubst(Cond,Subst,(In,Out),(emptySet,emptySet))] := H
    if VALID(Cond => L === null) .

  eq derive((stackInGraph(In,Out)(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> cases(answer([L + 1, graph . switch, 0] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                     ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . marked, 1),
              answer([L + 1, graph . switch, 0] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                     ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . marked, 1),
              answer([L + 1, graph . switch, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(n F)
                     ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(F) ** H ** #(n n n n F), graph . marked, 1))) .

  eq derive((stackInGraph(In,Out)(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> cases(answer([L, graph . marked, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                     ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . switch, 0),
              answer([L, graph . marked, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                     ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . switch, 0),
              answer([L, graph . marked, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(n F)
                     ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(F) ** H ** #(n n n n F), graph . switch, 1))) .

  eq derive((stackInGraph(In,Out)(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> cases(answer([L, graph . marked, 1] ** [L + 1, graph . switch, 0] ** [L + 3, graph . right, n F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                     ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . left, F),
              answer([L, graph . marked, 1] ** [L + 1, graph . switch, 0] ** [L + 3, graph . right, n F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                     ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . left, F),
              answer([L, graph . marked, 1] ** [L + 1, graph . switch, 1] ** [L + 3, graph . right, n F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(n F)
                     ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(F) ** H ** #(n n n n F), graph . left, F))) .

  eq derive((stackInGraph(In,Out)(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> cases(answer([L, graph . marked, 1] ** [L + 1, graph . switch, 0] ** [L + 2, graph . left, F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                     ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . right, n F),
              answer([L, graph . marked, 1] ** [L + 1, graph . switch, 0] ** [L + 2, graph . left, F]
                     ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                     ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . right, n F),
              answer([L, graph . marked, 1] ** [L + 1, graph . switch, 1] ** [L + 2, graph . left, F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(n F)
                    ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(F) ** H ** #(n n n n F), graph . right, n F))) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                            ** stackInGraph(In1,Out1)(D1') ** cleanGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION In2 UNION oneElemSet(L), Out1 UNION Out2 MINUS In1 UNION In2 UNION oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
       and VALID(Cond => D1 === D1') and VALID(Cond => D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                            ** stackInGraph(In1,Out1)(D1')
  => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION oneElemSet(L), Out1 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D1 === D1')
       and (VALID(Cond => D2 === null) or VALID(Cond => oneElemSet(D2) IN In1) or VALID(Cond => D2 === L) or VALID(Cond => oneElemSet(D2) IN Out1)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                            ** cleanGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** stackInGraph(In2 UNION oneElemSet(L), Out2 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D2 === D2')
       and (VALID(Cond => D1 === null) or VALID(Cond => oneElemSet(D1) IN In2) or VALID(Cond => D1 === L) or VALID(Cond => oneElemSet(D1) IN Out2)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
  => (Cond,Subst) |- H' := H ** stackInGraph(oneElemSet(L), emptySet)(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
       and (VALID(Cond => D1 === null) or VALID(Cond => D1 === L)) and (VALID(Cond => D2 === null) or VALID(Cond => D2 === L)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                            ** stackInGraph(In1,Out1)(D1') ** markedGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION In2 UNION oneElemSet(L), Out1 UNION Out2 MINUS In1 UNION In2 UNION oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D1 === D1')
       and VALID(Cond => D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                            ** markedGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** stackInGraph(In2 UNION oneElemSet(L), Out2 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D2 === D2')
       and (VALID(Cond => D1 === null) or VALID(Cond => oneElemSet(D1) IN In2) or VALID(Cond => D1 === L) or VALID(Cond => oneElemSet(D1) IN Out2)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                            ** markedGraph(In1,Out1)(D1') ** stackInGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION In2 UNION oneElemSet(L), Out1 UNION Out2 MINUS In1 UNION In2 UNION oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
       and VALID(Cond => D1 === D1') and VALID(Cond => D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                            ** markedGraph(In1,Out1)(D1')
  => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION oneElemSet(L), Out1 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D1 === D1')
       and (VALID(Cond => D2 === null) or VALID(Cond => oneElemSet(D2) IN In1) or VALID(Cond => D2 === L) or VALID(Cond => oneElemSet(D2) IN Out1)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                            ** stackInGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** stackInGraph(In2 UNION oneElemSet(L), Out2 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D2 === D2')
       and (VALID(Cond => D1 === null) or VALID(Cond => oneElemSet(D1) IN In2) or VALID(Cond => D1 === L) or VALID(Cond => oneElemSet(D1) IN Out2)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
  => (Cond,Subst) |- H' := H ** stackInGraph(oneElemSet(L), emptySet)(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
       and (VALID(Cond => D1 === null) or VALID(Cond => D1 === L)) and (VALID(Cond => D2 === null) or VALID(Cond => D2 === L)) .



-------------------------------
--- Schorr-Waite with graph ---
-------------------------------

  op nodes : -> Var .
  ops ?nodes1 ?nodes2 ?nodes3 ?out1 ?out2 : -> FreshVar .


eq Schorr-Waite-graph =
    //@ assume [cleanGraph(nodes,emptySet)(root)] ;
    t = root ;
    p = null ;
//@ inv t == null && [stackInGraph(nodes,emptySet)(p) ** ?rest]
        || t != null &&
           (* t == 1 &&& [markedGraph(?nodes1,?out1)(t) ** stackInGraph(?nodes2,?out2)(p) ** ?rest] ||
            * t == 0 &&& [ cleanGraph(?nodes1,?out1)(t) ** stackInGraph(?nodes2,?out2)(p) ** ?rest])
            && nodes == ?nodes1 UNION ?nodes2 && ?out1 in ?nodes2 && ?out2 in ?nodes1
    while (p != null || (t != null && * t == 0)) {
      if (t == null || * t == 1) {
        if (*(p + 1) == 1) {       --- POP
          q = t ;                  --- q = t
          t = p ;                  --- t = p
          p = *(p + 3) ;           --- p = p -> right
          *(t + 3) = q ;           --- t -> right = q
        }
        else {                     --- SWING
          q = t ;                  --- q = t
          t = *(p + 3) ;           --- t = p -> right
          *(p + 3) = *(p + 2) ;    --- p -> right = p -> left
          *(p + 2) = q ;           --- p -> left = q
          *(p + 1) = 1 ;           --- p -> c = 1
        }
      }
      else {                       --- PUSH
        q = p ;                    --- q = p
        p = t ;                    --- p = t
        t = *(t + 2) ;             --- t = t -> left
        *(p + 2) = q ;             --- p -> left = q
        * p = 1 ;                  --- p -> m = 1
        *(p + 1) = 0 ;             --- p -> c = 0
      }
    } ;
    //@ assert [markedGraph(nodes,emptySet)(t)]
.

***)

***(
sort Graph .
op nodes : Graph -> Var .
op edges : Graph -> K .

op  cleanGraph[_] : Graph -> HeapDefName [prec 0] .
op markedGraph[_] : Graph -> HeapDefName [prec 0] .
op stackInGraph[_] : Graph -> HeapDefName [prec 0] .

op graph : -> Graph .

op graph[_] : Graph -> HeapDefName [prec 0] .

var G : Graph .  var Kl : KList .

op ?in : -> FreshVar .

eq cleanGraph[G](Kl) = cleanGraph[G](nodes(G))(Kl) .
eq markedGraph[G](Kl) = markedGraph[G](nodes(G))(Kl) .
eq stackInGraph[G](Kl) = stackInGraph[G](nodes(G))(Kl) .

op {_,_,_} : K LocType K -> K .


----------------------------
--- cleanGraph[G](In)(L) ---
----------------------------
  eq //@ assumeCond(cleanGraph[G](In)(L)) = cases(//@ Assume(In === emptySet), //@ Assume(oneElemSet(L) IN In /\ ~(L === null))) .

 ceq (Cond,Subst) |- H' := H ** cleanGraph[G](In)(L)
   = (Cond /\ In === emptySet, Subst) |- H' := H if VALID(Cond => L === null) .
 ceq (Cond,Subst) |- H' ** cleanGraph[G](In)(L) := H
   = (Cond,(Subst,checkAndMakeSubst(Cond,Subst,In,emptySet))) |- H'[checkAndMakeSubst(Cond,Subst,In,emptySet)] := H  if VALID(Cond => L === null) .

  eq derive((cleanGraph[G](In)(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L + 1, graph[G] . switch, undef] ** [L + 2, graph[G] . left, F] ** [L + 3, graph[G] . right, n F]
               ** cleanGraph[G](n n F)(F) ** cleanGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . marked, 0)) .

  eq derive((cleanGraph[G](In)(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L, graph[G] . marked, 0] ** [L + 2, graph[G] . left, F] ** [L + 3, graph[G] . right, n F]
               ** cleanGraph[G](n n F)(F) ** cleanGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . switch, undef)) .

  eq derive((cleanGraph[G](In)(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L, graph[G] . marked, 0] ** [L + 1, graph[G] . switch, undef] ** [L + 3, graph[G] . right, n F]
            ** cleanGraph[G](n n F)(F) ** cleanGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . left, F)) .

  eq derive((cleanGraph[G](In)(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L, graph[G] . marked, 0] ** [L + 1, graph[G] . switch, undef] ** [L + 2, graph[G] . left, F]
            ** cleanGraph[G](n n F)(F) ** cleanGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . right, n F)) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 0] ** [L1, graph[G] . switch, undef] ** [L2, graph[G] . left, D1]
                           ** [L3, graph[G] . right, D2] ** cleanGraph[G](In1)(D1') ** cleanGraph[G](In2)(D2')
  => (Cond,Subst) |- H' := H ** cleanGraph[G](oneElemSet(L) UNION In1 UNION In2)(L)
    if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === D2'
                 /\ oneElemSet({L, left, D1'}) IN edges(G) /\ oneElemSet({L, right, D2'}) IN edges(G) /\ In1 INTERSECT In2 === emptySet) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 0] ** [L1, graph[G] . switch, undef] ** [L2, graph[G] . left, D1]
                           ** [L3, graph[G] . right, D2] ** cleanGraph[G](In1)(D1')
  => (Cond,Subst) |- H' := H ** cleanGraph[G](oneElemSet(L) UNION In1)(L)
    if VALID(Cond => (D2 === null \/ D2 === L \/ oneElemSet(D2) IN In1) /\ L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1'
                 /\ oneElemSet({L, left, D1'}) IN edges(G) /\ oneElemSet({L, right, D2}) IN edges(G)) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 0] ** [L1, graph[G] . switch, undef] ** [L2, graph[G] . left, D1]
                           ** [L3, graph[G] . right, D2] ** cleanGraph[G](In2)(D2')
  => (Cond,Subst) |- H' := H ** cleanGraph[G](oneElemSet(L) UNION In2)(L)
    if VALID(Cond => (D1 === null \/ D1 === L \/ oneElemSet(D1) IN In2) /\ L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D2 === D2'
                 /\ oneElemSet({L, left, D1}) IN edges(G) /\ oneElemSet({L, right, D2'}) IN edges(G)) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 0] ** [L1, graph[G] . switch, undef] ** [L2, graph[G] . left, D1] ** [L3, graph[G] . right, D2]
  => (Cond,Subst) |- H' := H ** cleanGraph[G](oneElemSet(L))(L)
    if VALID(Cond => (D1 === null \/ D1 === L) /\ (D2 === null \/ D2 === L) /\ L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3
                 /\ oneElemSet({L, left, D1}) IN edges(G) /\ oneElemSet({L, right, D2}) IN edges(G)) .



-----------------------------
--- markedGraph[G](In)(L) ---
-----------------------------
  eq //@ assumeCond(markedGraph[G](In)(L)) = cases(//@ Assume(In === emptySet), //@ Assume(oneElemSet(L) IN In /\ ~(L === null))) .

 ceq (Cond,Subst) |- H' := H ** markedGraph[G](In)(L) = (Cond /\ In === emptySet, Subst) |- H' := H if VALID(Cond => L === null) .
 ceq (Cond,Subst) |- H' ** markedGraph[G](In)(L) := H
   = (Cond,(Subst,checkAndMakeSubst(Cond,Subst,In,emptySet))) |- H'[checkAndMakeSubst(Cond,Subst,In,emptySet)] := H  if VALID(Cond => L === null) .

  eq derive((markedGraph[G](In)(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L + 1, graph[G] . switch, 1] ** [L + 2, graph[G] . left, F] ** [L + 3, graph[G] . right, n F]
            ** markedGraph[G](n n F)(F) ** markedGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . marked, 1)) .

  eq derive((markedGraph[G](In)(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L, graph[G] . marked, 1] ** [L + 2, graph[G] . left, F] ** [L + 3, graph[G] . right, n F]
            ** markedGraph[G](n n F)(F) ** markedGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . switch, 1)) .

  eq derive((markedGraph[G](In)(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L, graph[G] . marked, 1] ** [L + 1, graph[G] . switch, 1] ** [L + 3, graph[G] . right, n F]
            ** markedGraph[G](n n F)(F) ** markedGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . left, F)) .

  eq derive((markedGraph[G](In)(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L, graph[G] . marked, 1] ** [L + 1, graph[G] . switch, 1] ** [L + 2, graph[G] . left, F]
            ** markedGraph[G](n n F)(F) ** markedGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . right, n F)) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 1] ** [L1, graph[G] . switch, 1] ** [L2, graph[G] . left, D1]
                           ** [L3, graph[G] . right, D2] ** markedGraph[G](In1)(D1') ** markedGraph[G](In2)(D2')
  => (Cond,Subst) |- H' := H ** markedGraph[G](oneElemSet(L) UNION In1 UNION In2)(L)
    if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === D2'
                 /\ oneElemSet({L, left, D1'}) IN edges(G) /\ oneElemSet({L, right, D2'}) IN edges(G) /\ In1 INTERSECT In2 === emptySet) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 1] ** [L1, graph[G] . switch, 1] ** [L2, graph[G] . left, D1]
                           ** [L3, graph[G] . right, D2] ** markedGraph[G](In1)(D1')
  => (Cond,Subst) |- H' := H ** markedGraph[G](oneElemSet(L) UNION In1)(L)
    if VALID(Cond => (D2 === null \/ D2 === L \/ oneElemSet(D2) IN In1) /\ L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1'
                  /\ oneElemSet({L, left, D1'}) IN edges(G) /\ oneElemSet({L, right, D2}) IN edges(G)) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 1] ** [L1, graph[G] . switch, 1] ** [L2, graph[G] . left, D1]
                           ** [L3, graph[G] . right, D2] ** markedGraph[G](In2)(D2')
  => (Cond,Subst) |- H' := H ** markedGraph[G](oneElemSet(L) UNION In2)(L)
    if VALID(Cond => (D1 === null \/ D1 === L \/ oneElemSet(D1) IN In2) /\ L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D2 === D2'
                  /\ oneElemSet({L, left, D1}) IN edges(G) /\ oneElemSet({L, right, D2'}) IN edges(G)) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 1] ** [L1, graph[G] . switch, 1] ** [L2, graph[G] . left, D1] ** [L3, graph[G] . right, D2]
  => (Cond,Subst) |- H' := H ** markedGraph[G](oneElemSet(L))(L)
    if VALID(Cond => (D1 === null \/ D1 === L) /\ (D2 === null \/ D2 === L) /\ L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3
                  /\ oneElemSet({L, left, D1}) IN edges(G) /\ oneElemSet({L, right, D2}) IN edges(G)) .




------------------------------
--- stackInGraph[G](In)(L) ---
------------------------------
  eq //@ assumeCond(stackInGraph[G](In)(L)) = cases(//@ Assume(In === emptySet), //@ Assume(oneElemSet(L) IN In /\ ~(L === null))) .

 ceq (Cond,Subst) |- H' := H ** stackInGraph[G](In)(L) = (Cond /\ In === emptySet, Subst) |- H' := H if VALID(Cond => L === null) .
 ceq (Cond,Subst) |- H' ** stackInGraph[G](In)(L) := H
   = (Cond,(Subst,checkAndMakeSubst(Cond,Subst,In,emptySet))) |- H'[checkAndMakeSubst(Cond,Subst,In,emptySet)] := H  if VALID(Cond => L === null) .


  eq derive((stackInGraph[G](In)(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L + 1, graph[G] . switch, 1] ** [L + 2, graph[G] . left, F] ** [L + 3, graph[G] . right, n F]
            ** markedGraph[G](n n F)(F) ** markedGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . marked, 1)) .

  eq derive((stackInGraph(In,Out)(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In) ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F))
      ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> cases(answer([L + 1, graph . switch, 0] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                  ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                  ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . marked, 1),
           answer([L + 1, graph . switch, 0] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                  ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                  ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . marked, 1),
           answer([L + 1, graph . switch, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                  ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(n F)
                  ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(F) ** H ** #(n n n n F), graph . marked, 1))) .

  eq derive((markedGraph[G](In)(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L, graph[G] . marked, 1] ** [L + 2, graph[G] . left, F] ** [L + 3, graph[G] . right, n F]
            ** markedGraph[G](n n F)(F) ** markedGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . switch, 1)) .

  eq derive((markedGraph[G](In)(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L, graph[G] . marked, 1] ** [L + 1, graph[G] . switch, 1] ** [L + 3, graph[G] . right, n F]
            ** markedGraph[G](n n F)(F) ** markedGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . left, F)) .

  eq derive((markedGraph[G](In)(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null))
      ~> //@ Assume(oneElemSet(L) IN In /\ In IN nodes(G) /\ oneElemSet({L, left, F}) IN edges(G) /\ oneElemSet({L, right, n F}) IN edges(G))
     ~> answer([L, graph[G] . marked, 1] ** [L + 1, graph[G] . switch, 1] ** [L + 2, graph[G] . left, F]
            ** markedGraph[G](n n F)(F) ** markedGraph[G]((In MINUS (n n F)) MINUS oneElemSet(L))(n F) ** H ** #(n n n F), graph[G] . right, n F)) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 1] ** [L1, graph[G] . switch, 1] ** [L2, graph[G] . left, D1]
                           ** [L3, graph[G] . right, D2] ** markedGraph[G](In1)(D1') ** markedGraph[G](In2)(D2')
  => (Cond,Subst) |- H' := H ** markedGraph[G](oneElemSet(L) UNION In1 UNION In2)(L)
    if VALID(Cond => L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1' /\ D2 === D2'
                  /\ oneElemSet({L, left, D1'}) IN edges(G) /\ oneElemSet({L, right, D2'}) IN edges(G) /\ In1 INTERSECT In2 === emptySet) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 1] ** [L1, graph[G] . switch, 1] ** [L2, graph[G] . left, D1]
                           ** [L3, graph[G] . right, D2] ** markedGraph[G](In1)(D1')
  => (Cond,Subst) |- H' := H ** markedGraph[G](oneElemSet(L) UNION In1)(L)
    if VALID(Cond => (D2 === null \/ D2 === L \/ oneElemSet(D2) IN In1) /\ L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D1 === D1'
                  /\ oneElemSet({L, left, D1'}) IN edges(G) /\ oneElemSet({L, right, D2}) IN edges(G)) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 1] ** [L1, graph[G] . switch, 1] ** [L2, graph[G] . left, D1]
                           ** [L3, graph[G] . right, D2] ** markedGraph[G](In2)(D2')
  => (Cond,Subst) |- H' := H ** markedGraph[G](oneElemSet(L) UNION In2)(L)
    if VALID(Cond => (D1 === null \/ D1 === L \/ oneElemSet(D1) IN In2) /\ L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3 /\ D2 === D2'
                    /\ oneElemSet({L, left, D1}) IN edges(G) /\ oneElemSet({L, right, D2'}) IN edges(G)) .

 crl (Cond,Subst) |- H' := H ** [L, graph[G] . marked, 1] ** [L1, graph[G] . switch, 1] ** [L2, graph[G] . left, D1] ** [L3, graph[G] . right, D2]
  => (Cond,Subst) |- H' := H ** markedGraph[G](oneElemSet(L))(L)
    if VALID(Cond => (D1 === null \/ D1 === L) /\ (D2 === null \/ D2 === L) /\ L1 === L + 1 /\ L2 === L + 2 /\ L3 === L + 3
                    /\ oneElemSet({L, left, D1}) IN edges(G) /\ oneElemSet({L, right, D2}) IN edges(G)) .



----------------------
--- stackInGraph(L) ---
----------------------


  eq derive((stackInGraph(In,Out)(L) ** H ** #(F)), L)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> cases(answer([L + 1, graph . switch, 0] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                    ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . marked, 1),
             answer([L + 1, graph . switch, 0] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                    ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . marked, 1),
             answer([L + 1, graph . switch, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(n F)
                    ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(F) ** H ** #(n n n n F), graph . marked, 1))) .

  eq derive((stackInGraph(In,Out)(L) ** H ** #(F)), L + 1)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> cases(answer([L, graph . marked, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                    ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . switch, 0),
             answer([L, graph . marked, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                    ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . switch, 0),
             answer([L, graph . marked, 1] ** [L + 2, graph . left, F] ** [L + 3, graph . right, n F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(n F)
                    ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(F) ** H ** #(n n n n F), graph . switch, 1))) .

  eq derive((stackInGraph(In,Out)(L) ** H ** #(F)), L + 2)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> cases(answer([L, graph . marked, 1] ** [L + 1, graph . switch, 0] ** [L + 3, graph . right, n F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                    ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . left, F),
             answer([L, graph . marked, 1] ** [L + 1, graph . switch, 0] ** [L + 3, graph . right, n F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                    ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . left, F),
             answer([L, graph . marked, 1] ** [L + 1, graph . switch, 1] ** [L + 3, graph . right, n F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(n F)
                    ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(F) ** H ** #(n n n n F), graph . left, F))) .

  eq derive((stackInGraph(In,Out)(L) ** H ** #(F)), L + 3)
   = (//@ Assert(~(L === null)) ~> //@ Assume(oneElemSet(L) IN In)
      ~> //@ Assume(In MINUS oneElemSet(L) === (n n F) UNION (n n n F)) ~> //@ Assume((n n F) INTERSECT (n n n F) === emptySet)
     ~> cases(answer([L, graph . marked, 1] ** [L + 1, graph . switch, 0] ** [L + 2, graph . left, F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                    ** cleanGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . right, n F),
             answer([L, graph . marked, 1] ** [L + 1, graph . switch, 0] ** [L + 2, graph . left, F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(F)
                    ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(n F) ** H ** #(n n n n F), graph . right, n F),
             answer([L, graph . marked, 1] ** [L + 1, graph . switch, 1] ** [L + 2, graph . left, F]
                    ** stackInGraph((n n F), Out UNION (n n n F) UNION oneElemSet(L))(n F)
                    ** markedGraph((n n n F), Out UNION (n n F) UNION oneElemSet(L))(F) ** H ** #(n n n n F), graph . right, n F))) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                           ** stackInGraph(In1,Out1)(D1') ** cleanGraph(In2,Out2)(D2')
  => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION In2 UNION oneElemSet(L), Out1 UNION Out2 MINUS In1 UNION In2 UNION oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
       and VALID(Cond => D1 === D1') and VALID(Cond => D2 === D2') .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                           ** stackInGraph(In1,Out1)(D1')
  => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION oneElemSet(L), Out1 MINUS oneElemSet(L))(L)
    if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D1 === D1')
       and (VALID(Cond => D2 === null) or VALID(Cond => oneElemSet(D2) IN In1) or VALID(Cond => D2 === L) or VALID(Cond => oneElemSet(D2) IN Out1)) .

 crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
```

```
                             ** cleanGraph(In2,Out2)(D2')
   => (Cond,Subst) |- H' := H ** stackInGraph(In2 UNION oneElemSet(L), Out2 MINUS oneElemSet(L))(L)
     if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D2 === D2')
        and (VALID(Cond => D1 === null) or VALID(Cond => oneElemSet(D1) IN In2) or VALID(Cond => D1 === L) or VALID(Cond => oneElemSet(D1) IN Out2)) .

  crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
   => (Cond,Subst) |- H' := H ** stackInGraph(oneElemSet(L), emptySet)(L)
     if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
        and (VALID(Cond => D1 === null) or VALID(Cond => D1 === L)) and (VALID(Cond => D2 === null) or VALID(Cond => D2 === L)) .

  crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                             ** stackInGraph(In1,Out1)(D1') ** markedGraph(In2,Out2)(D2')
   => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION In2 UNION oneElemSet(L), Out1 UNION Out2 MINUS In1 UNION In2 UNION oneElemSet(L))(L)
     if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
        and VALID(Cond => D1 === D1') and VALID(Cond => D2 === D2') .

  crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 0] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                             ** markedGraph(In2,Out2)(D2')
   => (Cond,Subst) |- H' := H ** stackInGraph(In2 UNION oneElemSet(L), Out2 MINUS oneElemSet(L))(L)
     if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D2 === D2')
        and (VALID(Cond => D1 === null) or VALID(Cond => oneElemSet(D1) IN In2) or VALID(Cond => D1 === L) or VALID(Cond => oneElemSet(D1) IN Out2)) .

  crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                             ** markedGraph(In1,Out1)(D1') ** stackInGraph(In2,Out2)(D2')
   => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION In2 UNION oneElemSet(L), Out1 UNION Out2 MINUS In1 UNION In2 UNION oneElemSet(L))(L)
     if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
        and VALID(Cond => D1 === D1') and VALID(Cond => D2 === D2') .

  crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1]
                             ** [L3, graph . right, D2] ** markedGraph(In1,Out1)(D1')
   => (Cond,Subst) |- H' := H ** stackInGraph(In1 UNION oneElemSet(L), Out1 MINUS oneElemSet(L))(L)
     if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D1 === D1')
        and (VALID(Cond => D2 === null) or VALID(Cond => oneElemSet(D2) IN In1) or VALID(Cond => D2 === L) or VALID(Cond => oneElemSet(D2) IN Out1)) .

  crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
                             ** stackInGraph(In2,Out2)(D2')
   => (Cond,Subst) |- H' := H ** stackInGraph(In2 UNION oneElemSet(L), Out2 MINUS oneElemSet(L))(L)
     if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3) and VALID(Cond => D2 === D2')
        and (VALID(Cond => D1 === null) or VALID(Cond => oneElemSet(D1) IN In2) or VALID(Cond => D1 === L) or VALID(Cond => oneElemSet(D1) IN Out2)) .

  crl (Cond,Subst) |- H' := H ** [L, graph . marked, 1] ** [L1, graph . switch, 1] ** [L2, graph . left, D1] ** [L3, graph . right, D2]
   => (Cond,Subst) |- H' := H ** stackInGraph(oneElemSet(L), emptySet)(L)
     if VALID(Cond => L1 === L + 1) and VALID(Cond => L2 === L + 2) and VALID(Cond => L3 === L + 3)
        and (VALID(Cond => D1 === null) or VALID(Cond => D1 === L)) and (VALID(Cond => D2 === null) or VALID(Cond => D2 === L)) .



 op Schorr-Waite-graph-Complete : -> K .
 eq Schorr-Waite-graph-Complete =
 *** graph * schorr-waite(graph * root)
 *** requires [ cleanGraph[graph](root)] ;
 *** ensures  [markedGraph[graph](result)] ;
     //@ assume [cleanGraph[graph](root)] ;
     t = root ;
     p = null ;
  //@ assert t == null && [stackInGraph[graph](p) ** ?rest]
         || [ (cleanGraph[graph])(?in)(t) ** stackInGraph[graph](nodes(graph) MINUS ?in)(p) ** ?rest]
         || [markedGraph[graph](?in)(t) ** stackInGraph[graph](nodes(graph) MINUS ?in)(p) ** ?rest] ;
 stop ;
 //@ inv t == null && [stackInGraph[graph](p) ** ?rest]
         || [ (cleanGraph[graph])(?in)(t) ** stackInGraph[graph](nodes(graph) MINUS ?in)(p) ** ?rest]
         || [markedGraph[graph](?in)(t) ** stackInGraph[graph](nodes(graph) MINUS ?in)(p) ** ?rest]
     while (p != null || (t != null && * t == 0)) {
       if (t == null || * t == 1) {
         if (*(p + 1) == 1) {       --- POP
           q = t ;                  --- q = t
           t = p ;                  --- t = p
           p = *(p + 3) ;           --- p = p -> right
           *(t + 3) = q ;           --- t -> right = q
         }
         else {                     --- SWING
           q = t ;                  --- q = t
           t = *(p + 3) ;           --- t = p -> right
           *(p + 3) = *(p + 2) ;    --- p -> right = p -> left
           *(p + 2) = q ;           --- p -> left = q
           *(p + 1) = 1 ;           --- p -> c = 1
         }
       }
       else {                       --- PUSH
```

```
        q = p ;                    --- q = p
        p = t ;                    --- p = t
        t = *(t + 2) ;             --- t = t -> left
        *(p + 2) = q ;             --- p -> left = q
        * p = 1 ;                  --- p -> m = 1
        *(p + 1) = 0 ;             --- p -> c = 0
      }
    } ;
--- return(t)
    result = t ;
    //@ assert [markedGraph[graph](result)]
.
***)


endm
---rew [| Schorr-Waite-graph |] .
---q

--- simple test for # of paths
rew [| pgm1 |] .
---rew [| pgm2 |] .  --- slow
--- lists
rew [| wrong |] .
rew [| addHead |] .
rew [| addHeadComplete |] .
rew [| allocAndAddHead |] .
rew [| allocAndAddHeadComplete |] .
rew [| appendRec1 |] .
rew [| appendRec1Complete |] .
rew [| appendWhile1 |] .
rew [| appendWhile1Complete |] .
rew [| appendRec2 |] .
rew [| appendRec2Complete |] .
rew [| appendWhile2 |] .
rew [| appendWhile2Complete |] .
rew [| reverse1 |] .
rew [| reverse1Complete |] .
rew [| reverse2 |] .
rew [| reverse2Complete |] .
rew [| reverseWhile |] .
rew [| reverseWhileComplete |] .
rew [| disposeList |] .
rew [| disposeListWhile |] .
--- queues
rew [| enqueue |] .
rew [| enqueueComplete |] .
rew [| dequeue |] .
rew [| dequeueComplete |] .
rew [| transferOwner1 |] .
rew [| transferOwner1Complete |] .
rew [| transferOwner2 |] .
rew [| transferOwner2Complete |] .
rew [| stealQueue |] .
rew [| stealQueueComplete |] .
--- trees
rew [| allocTree |] .
rew [| allocTreeComplete |] .
rew [| mirror |] .
rew [| mirrorComplete |] .

---rew [| treeToList1 |] .
---rew [| treeToList2 |] .
---rew [| treeToListWhile |] .
q
--- Schorr-Waite
rew [| Schorr-Waite-tree |] .

rew [| Schorr-Waite-graph |] .
q
***)

rew [| Schorr-Waite-graph-Complete |] .

q

--- we are going to soon allow expressing configuration equalities as follows:
--- [list(S)(L) ** H] =  L == null && S == nil && [H] || L != null && [[L, (V,L')] ** list(S')(L') ** H] && S == V :: S'
--- [graph[B](G)(L) ** H] =  L == null && G == empty && [H]
```

```
|| L != null && [[L, (B,L1,L2)] ** H] && G == (L -> (L1,L2)) && {L1,L2} IN {L,null}
|| L != null && [[L, (B,L1,L2)] ** graph[B](G1)(L1) ** H] && G == G1 UNION (L -> (L1,L2)) && {L2} IN nodes(G) UNION {null}
|| L != null && [[L, (B,L1,L2)] ** graph[B](G2)(L2) ** H] && G == G2 UNION (L -> (L1,L2)) && {L1} IN nodes(G) UNION {null}
|| L != null && [[L, (B,L1,L2)] ** graph[B](G1)(L1) ** graph[B](G2)(L2) ** H] && G == G1 UNION G2 UNION (L -> (L1,L2))
```