Types with Semantics

Soundness Proof Assistant

Olha Shkaravska

Institut of Informatics, Ludwig-Maximilians University

shkarav@tcs.ifi.lmu.de

Abstract

We present a parametric Hoare-like logic for computer-aided reasoning about typeable properties of functional programs. The logic is based on the concept of a *specialised assertion*, which is a predicate expressing the semantics of a typing judgment in a logical framework (here higher-order logic). Replacing in a type inference rule the judgments by the appropriate specialised assertions, one obtains a *specialised rule*. Specialised assertions have a uniform format, and soundness proofs of specialised rules employ uniform sequences of steps for a variety of type systems. This allows to abstract from the type system and define *parametric specialised assertion and rules*. Moreover, we define a *parametric soundness condition* for each program construct which ensures soundness of the corresponding rule. To prove soundness of the specialised rule for the concrete type system one checks the condition and instantiates the parametric rule. We consider specialised logics for "non-pure" type systems which contain rules with semantical side conditions. A semantical condition makes type-checking for such a system undecidable. However, the condition may be approximated by another type system. We introduce a product-like composition of two inference systems that eliminates semantical side conditions. The soundness condition of the composition follows straightforwardly from the soundness conditions of its components and a statement about the approximation.

Categories and Subject Descriptors F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs—Assertions, Mechanical verification, Pre- and Post-conditions; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Denotational semantics; F.3.3 [*Logics and Meaning of Programs*]: Studies of Program Constructs—Control primitives, type structure; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Mechanical theorem proving; D.2.4 [*Software Engineering*]: Software/Program Verification—Assertion checkers, correctness proofs; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Automatic analysis of algorithms, program verification; I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving—Inference engines

General Terms Theory, Verification

Keywords Assertions, program logics, type systems, automated theorem proving

1. Introduction

Type systems and program logics such as Hoare logic tend to serve the same purpose: to ensure properties of programs. However, the big advantage of type systems is that they present program properties in a relatively simple form. Moreover, the inference rules of even some quite expressive type systems generate syntactically-driven algorithms which allow to prove the corresponding properties without human interference. This is a very desirable feature for the systems used in applied program verification, see, for example, [4, 12, 13].

Before using a type system in program verification one has the obligation to (ideally) mechanically justify its soundness w.r.t. the operational semantics of the program language under consideration. This is often not an easy exercise, but when it is done once, one has the type system correct "forever". A real problem with type systems is their compatibility. The syntactic nature of properties they express makes it difficult to compare their merits.

Program logics are, in general, more expressive than typing systems. However, despite the impressive progress in the area of automated theorem proving, proof processes do need human participation, for instance, to provide invariants. Yet another problem arises when a predicate to prove, say, stemming from a verification condition generator, is too complicated to be proved automatically, requiring non-trivial instantiations of quantifiers. We will consider an example of such statement later.

In this work we develop the idea of a synergy between type systems and program logics [5]. One may represent a type-inference rule in the form of a proof rule in the logic, see, for instance, [3]. In this way a type inference system generates a logic for which automated, syntactically driven proof scripting is possible. On the other side, the program logic helps to prove soundness of the type system and serves as a basis on which two different type system may be compared and combined.

The remainder of this paper is split into two main parts. In the overview we introduce concepts and highlight the ideas on which the current work is based. The next section is technical, devoted mainly to concrete issues of implementation. We finish the paper with a short overview of related papers, a summary of the results and plans for the future work.

2. Overview of Results

Working on the Mobile Resource Guarantees (MRG) project [1] we have developed a multi-layered infrastructure for certification of resources. We believe that such infrastructure may be used for reasoning about other program properties. We will sketch it briefly from bottom to top, see [11] for more detail.

$$\begin{tikzcd}
\textit{High-Level Type System} \quad \Gamma \vdash_{\Sigma} e^{HLL} : T \arrow[d, "\textit{compile}"] \\
\textit{Specialised Logic} \quad G_{\Sigma} \rhd e : D(\Gamma,\, T) \arrow[d, no head] \\
\textit{Basic Program Logic} \quad G \rhd e : S \arrow[d, no head, line width=2pt] \\
\textit{Operational Semantics} \quad E \vdash h, e \Downarrow (v, h')
\end{tikzcd}$$

Figure 1. A family of logics for resource consumption

At the basis we have our (trusted) *operational semantics*.

On the next level we have a general-purpose, *basic program logic* for partial correctness. Its role is to serve as a platform at which various higher level logics may be unified. The latter purpose makes logical completeness of the program logic a desirable property. Of course, soundness remains mandatory, as the trustworthiness of any application logic defined at higher levels depends upon it.

On top of the general-purpose logic, a *specialised logic* (for example, the heap logic of [3]) is defined to capture a particular property. This logic uses a restricted format of assertions, called *specialised assertions*, which reflects the information of the high-level type system.

Judgements in the specialised logic have the form $G_{\Sigma} \rhd e : D(\Gamma,\, T)$, where the expression e is the result of compiling a high-level term e^{HLL} down to a low-level language, and the information in the high-level type system is encoded in a special form of assertion that depends on the context and type associated to e^{HLL}, $D(\Gamma,\, T)$. G_{Σ} is a set of assumptions (invariants) corresponding to a first-order signature Σ. Depending on the property of interest, this level may be further refined into a hierarchy of proof systems, for example if parts of the soundness argument of the specialised assertions can be achieved by different type systems.

In contrast to the general-purpose logic, this specialised logic is not expected to be logically complete, but it should provide support for automated proof search. In the case of the logic for heap consumption, this is achieved by formulating a system of specialised assertions whose level of granularity is roughly similar to the high-level type system. However, the rules are expressed in terms of code fragments in the low-level language. Since the side conditions of the typing rules are computationally easy to validate, automated proof search is supported by the syntax-directness of the typing rules. At points where syntax-directness fails — such as recursive program structures — the necessary invariants are provided by the type system.

Thus, on the top level we find a *high-level type system*, that encodes information on resource consumption. In the judgement $\Gamma \vdash_{\Sigma} e^{HLL} : T$, the term e^{HLL} has an (extended) type T in a context Γ with the first-order signature Σ.

The aim of the present work is to ease the computer-aided verification of the soundness of specialised logics. The soundness proof, implemented in a theorem prover, justifies the usage of the logic in program verification.

A "smart" proof scripting assumes:

- modularisation of specialised assertions, which allows one to consider a complex property as a combination of simpler statements,
- modularisation of proofs itself, that is splitting proofs into lemmas,
- proof scripts reuse.

The latter becomes possible due to common features in structures of assertions and soundness proofs arising from various type systems. This allows to design a *parametric specialised logic*, based on the modularisation principles above.

In the rest of this paper we discuss the components of our reasoning infrastructure in more detail and the design of the parametric logic. We will start with a general outline and then continue with concrete logics for the low-level language used in the MRG project.

2.1 A basis for verification

We consider an impure functional language with its operational semantics formalised in a higher-order logic. The big-step operational semantics is a partial correctness relation of the form $E \vdash h, e \Downarrow (h', v)$ where e is an expression in the language, an environment (stack) $E : \mathit{Envs} \equiv \mathit{Vars} \to \mathit{Vals}$, heaps $h,\ h : \mathit{Heaps} \equiv \mathit{Fields} \to \mathit{Locs} \to \mathit{Vals}$ are maps from the sets of variable names, field names and locations respectively, into the set of values, and v is a return value. The relation means that if with the initial environment E and heap h the expression e terminates, then it evaluates to v and changes the heap to h'.

Program properties are expressed as partial correctness assertions of the form $G \rhd e : P$, where G is a set of assumptions for methods and functions (invariants) and P is a predicate of type

$$\mathit{Spec} \equiv \mathit{Envs} \to \mathit{Heaps} \to \mathit{Heaps} \to \mathit{Vals} \to \mathit{Bool}.$$

We call P a *specification* of the expression e.

Each program construct, such as a simple expression x or `let`-binding, has its strongest specification, S, that is the specification which mirrors its operational semantics, $S\, E\, h\, h'\, v \equiv E \vdash h, e \Downarrow (h', v)$. The rules for function calls and method invocations, at which we will have a closer look later in this paper, manage recursion. Any predicate which holds for a construct follows from its strongest specification. The set of the strongest specifications for the language constructs together with an axiom

$$\frac{(e,\ P) \in G}{G \rhd e : P}\textsc{VAx}$$

and the rule of consequence,

$$\frac{\begin{array}{c} G \rhd e : Q \\ \forall E\, h\, h'\, v. (Q\, E\, h\, h'\, v) \to (P\, E\, h\, h'\, v) \end{array}}{G \rhd e : P}\textsc{Conseq}$$

constitutes the *basic logic*, see [2].

The basic logic is sound and (relatively) complete w.r.t. the operational semantics. The strongest specification for an expression e is derivable applying the proof rules of the basic logics.

To express the soundness and completeness result we introduce a predicate of the semantical validity $\models e : P$, which means, that for any E, h, h', v, the relation $E \vdash h, e \Downarrow (v, h')$ entails $P\ E\, h\, h'\, v$. This is extended to contexts in the obvious way. The soundness of the logic means that for any set of assumption G, an expression e and a specification P the provability $G \rhd e : P$ implies semantical validity $G \models e : P$, see [2]. The completeness theorem ensures the converse, if $G = \emptyset$. We have extended the basic logic from [2] with an AND-rule:

$$\frac{G \rhd e :\ P \quad G \rhd e :\ Q}{G \rhd e :\ P \bigwedge Q}\textsc{And}$$

where $- \bigwedge - \ :\ \mathit{Spec} \to \mathit{Spec} \to \mathit{Spec}$ is defined as

$$(P \bigwedge Q)\ E\, h\, h'\, v\ \equiv\ P\ E\, h\, h'\, v \wedge Q\ E\, h\, h'\, v.$$

2.2 From type systems to specialised logics

Consider now a typing system for the language. Following the Foundational Proof Carrying Code (FPCC) principle "to check a checker" one may want to justify its use in a verification framework. In other words - to prove its soundness w.r.t. the operational semantics and intended meaning of the judgment.

A typing judgment $\Gamma\ \vdash_\Sigma\ e^{HLL}\ :\ T$, where Σ is a first-order signature, is a syntactical statement, for which the designer of the system assumes a certain meaning. The semantics of a judgment, possibly modulo compilation, is formalised in in the form of a *specialised assertion*. "Modulo compilation" means that the type system and program logic are not necessary defined for the same language. The type system may be defined for a high-level language, whereas the program logic is defined for the language where the high-level one is compiled to. In MRG we work with typing systems for the high-level resource-aware functional language `Gamelot` [7], which is compiled into a low-level `Grail` [8]. The program logic is defined for `Grail` [2].

Consider a simple property of "well-formed datatypes". To model datatypes, we use a predicate $h\ \Vdash_T\ a$, expressing that an address a in heap h is the start of a (high-level) data-type T. In the basic program logic we can express the fact that the a method $c.m_f$ compiled from the high-level function $f\ ::\ \mathit{List}(T)\ \to\ \mathit{List}(T)$ preserves a well-formed data-type as follows:

$$\rhd\ c.m_f(x) :\ \lambda E\ h\ h'\ v.\ h \Vdash_{\mathit{List}(T)}\ E\langle x\rangle\ \longrightarrow\ h' \Vdash_{\mathit{List}(T)}\ v$$

To abstract over the details of modeling data-structures, we construct a specialised logic for this property, by restricting the form of the assertions to reflect the high-level property to be formalised.

The definition of the specialised assertion carries with it the high-level types of the variables, and accesses low-level predicates, which are needed to express the property but should be hidden in the specialised logic itself. For instance,

$$\begin{array}{l} D\big(x : \mathit{List}(T), y : \mathit{List}, \quad \mathit{List}(T)\big) \equiv \lambda E\ h\ h'\ v\ p. \\ h \Vdash_{\mathit{List}(T)}\ E\langle x\rangle\ \wedge\ h \Vdash_{\mathit{List}(T)}\ E\langle y\rangle\ \longrightarrow \\ h' \Vdash_{\mathit{List}(T)}\ E\langle x\rangle\ \wedge\ h' \Vdash_{\mathit{List}(T)}\ E\langle y\rangle\ \wedge\ h' \Vdash_{\mathit{List}(T)}\ v \end{array}$$

Substituting in an inference rule the judgments with the specialised assertions one obtains *specialised* proof *rules*, which form a *specialised* program *logic* for the type system. To prove the soundness of an inference rule means to prove the soundness of the corresponding specialised rule in the higher-order logic, where the operational semantics and the basic logic are defined.

Usually, the soundness of a specialised rule is to be proven by a human operator. A verification framework, such as FPCC or its MRG-version, requires that the soundness proofs are implemented in a theorem prover rather than "on the paper". One may prove the soundness of the rule directly from the operational semantics of the appropriate construct or applying the rule of consequence and the basic rule for the construct. The subgoal which is to be proven after that is a challenging problem for many typing systems, especially for the `let`-construct. Correct modularisation of proof scripts substantially improves their quality and speeds up the work.

2.3 Parametric logic

As we have seen above, the meaning of a typing judgment may be formalised in the form of an implication containing a statement about the pre-state in the negative position and a predicate about the post-state in the conclusion. Let

$$\begin{array}{rcl} \mathit{PreSpec} & \equiv & \mathit{Envs} \to \mathit{Heaps} \to \mathit{Bool} \\ \mathit{PostSpec} & \equiv & \mathit{Heaps} \to \mathit{Vals} \to \mathit{Bool} \\ \mathit{PreSpec}(\alpha) & \equiv & \alpha \to \mathit{PreSpec} \\ \mathit{PostSpec}(\alpha) & \equiv & \alpha \to \mathit{PostSpec} \end{array}$$

The format under consideration is as follows: P in $G\ \rhd\ e\ :\ P$ is split into predicates

$$\begin{array}{rcl} P & \equiv & \mathit{Pre} \Rightarrow \mathit{Post} \\ (\mathit{Pre} \Rightarrow \mathit{Post})\ \ E\, h\, h'\, v & \equiv & \forall X.\, (\mathit{Pre}\ X\ E\, h) \longrightarrow (\mathit{Post}\ X\ h'\ v). \end{array}$$

where

$$\begin{array}{rcl} \mathit{Pre} & :: & \mathit{PreSpec}(\alpha) \\ \mathit{Post} & :: & \mathit{PostSpec}(\alpha) \\ - \Rightarrow - & :: & \mathit{PreSpec}(\alpha) \times \mathit{PostSpec}(\alpha) \to \mathit{Spec} \end{array}$$

The predicates Pre and Post and a type parameter α depend on the type system.

We want to design a *parametric specialised logic*, that is a set of specialised proof rules parameterised over Pre and Post predicates. For each program construct we define its soundness predicate over Pre and Post, such that assuming this predicate holds, the corresponding parametric rule is sound, that is provable in the basic logic. To prove soundness of a given type system T one must

- express the semantics of its typing judgment in the form of assertion $D(\Gamma,\ T) = \mathit{Pre}^\Gamma \Rightarrow \mathit{Post}^T$,
- for each program construct prove its soundness predicate and instantiate the corresponding parametric rule with Pre^Γ and Post^T.

In the next section we consider an example which shows how a modular soundness condition and proof of a parametric rule are designed.

2.4 Combining type systems

The parametric framework has yet another important aspect. It allows us to combine type systems as corresponding specialised logics on the base of the same operational semantics and basic logic. One can consider combinations of two kinds.

The first one is a conjunction of two specifications [1]

$$G \rhd e :\ \Big(\mathit{Pre}^1 \Rightarrow \mathit{Post}^1\ \bigwedge\ \mathit{Pre}^2 \Rightarrow \mathit{Post}^2\Big)$$

such that $G\ \rhd\ e\ :\ \mathit{Pre}^1\ \Rightarrow\ \mathit{Post}^1$ and $G\ \rhd\ e\ :\ \mathit{Pre}^2\ \Rightarrow\ \mathit{Post}^2$ are independently provable applying specialised rules of the appropriate logics. The soundness of such combination immediately follows from the soundness of its constituents.

In some cases, which we will discuss soon, one considers *interleaving specialised assertions*, that is, of the form

$$G \rhd e :\ \mathit{Pre}^1 \cdot \mathit{Pre}^2 \Rightarrow \mathit{Post}^1 \cdot\cdot \mathit{Post}^2.$$

The infix operators $\cdot$ and $\cdot\cdot$ are of types

$$\begin{array}{l} \mathit{PreSpec}(\alpha) \times \mathit{PreSpec}(\beta) \to \mathit{PreSpec}(\alpha \times \beta) \\ \mathit{PostSpec}(\alpha) \times \mathit{PostSpec}(\beta) \to \mathit{PostSpec}(\alpha \times \beta) \end{array}$$

and respectively defined by:

$$\begin{array}{r} (\mathit{Pre}^1 \cdot \mathit{Pre}^2)\ (X_1, X_2)\ E\, h \equiv (\mathit{Pre}^1\ X_1\ E\, h) \wedge (\mathit{Pre}^2\ X_2\ E\, h) \\ (\mathit{Post}^1 \cdot\cdot \mathit{Post}^2)\ (X_1, X_2)\ h'\, v \equiv (\mathit{Post}^1\ X_1\ h'\, v) \wedge (\mathit{Post}^2\ X_2\ h'\, v) \end{array}$$

[1] Everywhere in this paper uppercase indices are used to distinguish specialised logics.

To complete this definition, we define inference rules in the combined logic system. The way of combination will become clear once we explain the reason of introducing interleaving logics. In practice one may face with *non-pure* type systems, by which we mean systems with inference rules containing semantical, i.e. non-statically verifiable, side conditions. The heap-space-aware inference from [4] is such an example.

Motivated by this work we may consider a semantic `let`-rules:

$$\frac{G \rhd e_1 : \mathit{Pre}_1 \Rightarrow \mathit{Post}_1 \qquad G \rhd e_2 : \mathit{Pre}_2 \Rightarrow \mathit{Post}_2 \qquad \models e_1 : \mathcal{P}}{G \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mathit{Pre} \Rightarrow \mathit{Post}.}\ \mathrm{S1}$$

This rule may be proven, thanks to completeness, from a different rule with $G \rhd e_1 : \mathcal{P}$ instead of $\models e_1 : \mathcal{P}$:

$$\frac{G \rhd e_1 : \mathit{Pre}_1 \Rightarrow \mathit{Post}_1 \qquad G \rhd e_2 : \mathit{Pre}_2 \Rightarrow \mathit{Post}_2 \qquad G \rhd e_1 : \mathcal{P}}{G \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mathit{Pre} \Rightarrow \mathit{Post}.}\ \mathrm{S1'}$$

Syntactically driven proof scripting in such systems is not possible. In [4] the predicate $\mathcal{P}$ expresses the property of *benign sharing*, that is, the parts of heaps accessible from the reference variables of e_2 are not destroyed during evaluation of e_1. In this case $\mathcal{P}$ is a parametrical predicate in the set of variables in e_2.

However the side condition $G \rhd e_1 : \mathcal{P}$ may be eliminated if it can be approximated by another type system, such as

$$\frac{G' \rhd e_1 : \mathit{Pre}'_1 \Rightarrow \mathit{Post}'_1 \qquad G' \rhd e_2 : \mathit{Pre}'_2 \Rightarrow \mathit{Post}'_2}{G' \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mathit{Pre}' \Rightarrow \mathit{Post}'}\ \mathrm{S2}$$

with Pre', Pre'_1,Post'_1 implying $\mathcal{P}$. The combined rule must be free of the semantical condition:

$$\frac{G \times G' \rhd e_1 : \mathit{Pre}_1 \cdot \mathit{Pre}'_1 \Rightarrow \mathit{Post}_1 \cdot \cdot \mathit{Post}'_1 \qquad G \times G' \rhd e_2 : \mathit{Pre}_2 \cdot \mathit{Pre}'_2 \Rightarrow \mathit{Post}_2 \cdot \cdot \mathit{Post}'_2}{G \times G' \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mathit{Pre} \cdot \mathit{Pre}' \Rightarrow \mathit{Post} \cdot \cdot \mathit{Post}'}\ \mathrm{S{\small C}}$$

with the product of sets of invariants defined in the obvious way.

This gives a definition of the combined `let`-rule: SC is a rule in the combined system if and only if S1' and S2 are rules in the components. Other rules are combined in the similar way. We discuss the soundness issue of combined rules in the next section.

It is easy to see, that $G \times G' \rhd e : \mathit{Pre} \cdot \mathit{Pre}' \Rightarrow \mathit{Post} \cdot \cdot \mathit{Post}'$ is provable in the combined logic if and only if $G \rhd e : \mathit{Pre} \Rightarrow \mathit{Post}$ and $G' \rhd e : \mathit{Pre}' \Rightarrow \mathit{Post}'$ are provable in the corresponding logics. However, reduction to a conjunction of two assertion makes no sense since the type-inference for the first system alone is not possible.

3. The Language and its Logics

3.1 The Grail language

Grail is a special form of Java Virtual Machine Language (JVML) [8]. Grail retains the object and method structure, but it represents method bodies as sets of mutually tail-recursive first-order functions. Actual parameters in function calls coincide syntactically with the formal parameters of the function definitions. This allows function calls to be interpreted as immediate jump instructions of JVML. The syntax of expressions is defined by the grammar

$$\begin{array}{rcl} e \in \mathit{expr} & ::= & \texttt{null} \mid \texttt{int}\ i \mid \texttt{var}\ x \mid \texttt{prim}\ \mathit{op}\ x\ x \mid \\ & & \texttt{new}\ C\ [\overline{t_i := x_i}] \mid x.t \mid x.t{:=}x \mid \\ & & C.t \mid C.t{:=}x \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \\ & & e\,;\,e \mid \texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e \mid \\ & & \texttt{call}\ f \mid C.M(\overline{a}) \\ a \in \mathit{args} & ::= & \texttt{var}\ x \mid \texttt{null} \mid i \end{array}$$

where C, M, f, t, x range over class, method, function, field and variable names respectively, i ranges over integer constants, op denote a binary primitive operation, such as arithmetic for integers and comparisons over integers and heap references. Integers, references r and $\bot$ form the set of values Vals. Heap references are either `null` or of the form $\mathit{Ref}\ l$, where $l \in \mathit{Locs}$ is a location.

Expressions correspond to primitive sequences of byte-code instructions. For example, $x.t := y$ represents a dynamic field update. The binding `let` $x = e_1$ `in` e_2 is used if the evaluation of e_1 returns and integer or reference value on top of the JVML stack, $e_1;\ e_2$ if not. The instruction $C.M(\bar{a})$ represents static method invocation. See [8] for more detail about `Grail`.

3.2 The operational semantics and the basic logic

The program logic for `Grail` is based on the operational semantics $E \vdash h, e \Downarrow (h', v)$ via strongest specifications for each program construct. For instance, the strongest specification for expression `var` x is a parametric specification VAR of type $\mathit{Vars} \rightarrow \mathit{Spec}$ defined as follows:

$$\mathsf{VAR}\ x \ \equiv \lambda\, E\, h\, h'\, v.\, h' = h \wedge v = E\langle x\rangle$$

Similarly, the strongest conditions for dynamic field update, `if`-branching and `let`-binding have types

$$\begin{array}{rcl} \mathsf{PUTFI} & :: & \mathit{Vars} \rightarrow \mathit{Fields} \rightarrow \mathit{Vars} \rightarrow \mathit{Spec} \\ \mathsf{IF} & :: & \mathit{Vars} \rightarrow \mathit{Spec} \rightarrow \mathit{Spec} \rightarrow \mathit{Spec} \\ \mathsf{LET} & :: & \mathit{Vars} \rightarrow \mathit{Spec} \rightarrow \mathit{Spec} \rightarrow \mathit{Spec} \end{array}$$

respectively. They are defined as follows:

$$\begin{array}{ll} \mathsf{PUTFI}\ x\, t\, y \equiv \lambda\, E\, h\, h'\, v. & \exists\, l.\ E\langle x\rangle = \mathit{Ref}\ l\ \wedge \\ & h' = h.t(l := E\langle y\rangle) \wedge \\ & v = \bot \\ \mathsf{IF}\ x\, P\, Q \equiv \lambda\, E\, h\, h'\, v. & \big(E\langle x\rangle = \mathsf{true} \longrightarrow (P\, E\, h\, h\, v)\big) \wedge \\ & \big(E\langle x\rangle = \mathsf{false} \longrightarrow (Q\, E\, h\, h\, v)\big) \wedge \\ & \big(E\langle x\rangle = \mathsf{true} \vee E\langle x\rangle = \mathsf{false}\big) \\ \mathsf{LET}\ x\, P_1\, P_2 \equiv \lambda\, E\, h\, h'\, v. & \exists\, h_1\, w. \\ & (P_1\, E\, h\, h_1\, w)\ \wedge \\ & (w \neq \bot) \wedge \\ & (P_2\, E\langle x := w\rangle\, h_1\, h'\, v) \end{array}$$

The rules for the above program constructs look like that:

$$\frac{}{G \rhd \texttt{var}\ x : \mathsf{VAR}\ x}\ \mathrm{V{\small AR}}$$

$$\frac{}{G \rhd x.t{:=}y : \mathsf{PUTFI}\ x\, t\, y}\ \mathrm{P{\small UT}F{\small I}}$$

$$\frac{G \rhd e_t : P \quad G \rhd e_f : Q}{G \rhd \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \mathsf{IF}\ x\, P\, Q}\ \mathrm{I{\small F}}$$

$$\frac{G \rhd e_1 : P_1 \quad G \rhd e_2 : P_2}{G \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mathsf{LET}\ x\, P_1\, P_2}\ \mathrm{L{\small ET}}$$

The rule for a function call

$$\frac{G \cup \big(f(\bar{x}) : P\big) \rhd \mathsf{body}_f : \lambda\, E\, h\, h'\, v.\, P\, E\, h\, h'\, v}{G \rhd f(\bar{x}) : P}\ \mathrm{F{\small UN}}$$

allows one to recursively use the assumption that a call to f satisfies a specification when proving that the unfolded definition of f (body_f) indeed satisfies the specification. Because of the restrictions of `Grail`, described at the beginning of this section, the actual parameters $\bar{x} = (x_1, \ldots x_n)$ coincide with formal parameters as mentioned in body_f. For *method* invocations the appropriate rule instantiates parameters, substituting into the method body.

3.3 Specialised logics

We consider here the MRG's focus – the resource aware logic arising from the typing system of Hofmann and Jost [4].

Types in this system are annotated with natural numbers, such as $\mathit{List}(T, k)$, which means a list of type T, such that per each element there are k extra free heap units. We remark, that T is a annotated type as well, unless it is an un-boxed type, say `Unit` or `Int`.

A tying judgment $\Gamma,\, n \ \vdash_\Sigma \ e^{HLL} \ : \ T,\, n'$ may be read as: "under signature Σ, in typing context Γ and with n memory resources available, the high-level-language term e^{HLL} has type T with n' unused resources left over". Formally, judgments are defined via heap-space aware inference rules, such as:

$$\frac{n \geq 1 + k + n'}{\begin{array}{l}\Gamma,\, x_{hd} : T,\, x_t : \mathit{List}(T,\, k),\ n \vdash_\Sigma \\ \mathit{cons}(x_{hd},\, x_t) \ : \ \mathit{List}(T,\, k),\ n'\end{array}}\textsc{Cons}$$

and

$$\frac{\begin{array}{c}\Gamma_1,\, n \vdash_\Sigma \ e_1^{HLL} \ : \ T_0,\ n_0 \\ \Gamma_2,\, x : T_0,\ n_0 \vdash_\Sigma \ e_2^{HLL} \ : \ T,\ n\end{array}}{\Gamma_1,\, \Gamma_2,\, n \vdash_\Sigma \ \mathtt{let}\, x = e_1^{HLL}\ \mathtt{in}\ e_2^{HLL} \ : \ T,\ n}\textsc{Let}$$

We will represent the semantics of the typing judgment in the program logic as a specialized assertion $\lambda\, E\, h\, h'\, v.\ [\![U,\, n,\, \Gamma \ \blacktriangleright\ T,\, n']\!]$. To this aim, we need to define a notion of a *virtual cost* (or a *potential*). Following [3], we consider just the types unit $\mathbf{1}$, integers $\mathbf{I}$ and annotated lists of integers $\mathbf{L}(k)$, where k is a natural number. We define a relation $v \Vdash_T^h R,\, K$, which means that a stack value v, associated with a type T, points to a region R in a heap h and has a virtual cost K:

$$\frac{}{\bot \Vdash_{\mathbf{1}}^h \emptyset,\, 0}\mathrm{U} \qquad \frac{}{i \Vdash_{\mathbf{I}}^h \emptyset,\, 0}\mathrm{I}$$

$$\frac{}{\mathsf{null} \Vdash_{\mathbf{L}(k)}^h \emptyset,\, 0}\textsc{Nil} \qquad \frac{h.\ell.\mathsf{TL} \Vdash_{\mathbf{L}(k)}^h R,\, K}{\ell \Vdash_{\mathbf{L}(k)}^h \{\ell\} \uplus R,\, K + k}\textsc{Cons}$$

with TL for the tail-field and $\uplus$ denoting a disjoint union.

Let U denote a finite set of program variables. We extend the relation $\Vdash$ to U and a typing context Γ:

$$\frac{}{E,\, \emptyset \Vdash_\Gamma^h \emptyset,\, 0} \qquad \frac{E,\, U \Vdash_\Gamma^h R_1,\, K_1 \quad E\langle x\rangle \Vdash_{\Gamma(x)}^h R_2,\, K_2}{E,\, U \uplus \{x\} \Vdash_\Gamma^h R_1 \uplus R_2,\, K_1 + K_2}$$

According to the correctness theorem from [4], the high-level typing judgment $\Gamma,\, n \vdash e^{HLL} : T,\, n'$ translates into an assertion such that for any $E,\, h,\, h'\, v$, if the compiled expression e terminates on E and h with a value v and a heap h', then for any $q \geq 0$

- if the amount of free heap units (the free list size) before evaluation is $m \geq n + K + q$, where K is a virtual cost of variables from $U = \mathit{dom}\,\Gamma$, in the pre-state defined by E and h,
- then the amount of free heap units after the evaluation is $m' \geq n' + S + q$, where S is a virtual cost of the return value v, and q free heap units are left intact during the evaluation.

Note, that in the original typing system from [4] the disjointness condition is not necessary. All one needs is to ensure benign sharing. The latter can be approximated by linear typing.

The definition of the specialised assertion is as follows:

$$\begin{array}{l}[\![U,\, n,\, \Gamma \ \blacktriangleright\ T,\, n']\!] \equiv\ \forall\, q\, F\, R\, m\, K. \\ \left(\begin{array}{l}\mathit{freelist}(h,\, F,\, m) \wedge \\ E,\, U \Vdash_\Gamma^h R,\, K \wedge \\ R \cap F = \emptyset \wedge \\ m \geq n + K + q\end{array}\right) \longrightarrow \\ \left(\begin{array}{ll}\exists\, Q\, S\, m'\, H. & v \Vdash_T^h R,\, S \wedge \\ & \mathit{freelist}(h',\, H,\, m') \wedge \\ & Q \cap H = \emptyset \wedge Q \cup H \subseteq (R \cup F) \wedge \\ & \mathit{footrpint}(F \cup R,\, h,\, h') \wedge \\ & m' \geq n' + S + q \wedge \mathit{dom}\, h = \mathit{dom}\, h'\end{array}\right)\end{array}$$

The meaning of this assertion corresponds to the semantics of the corresponding high-level typing judgement. The interested reader may guess it her/himself or is referred to [3] for more detail. We have just reported it as an example of a statement whose presence in verification conditions makes them arguably difficult to prove automatically. Moreover, the proof of the soundness of the `let`-rule for such assertions is a technically difficult task even for a human operator. So, what can we do to cope with assertions of such complexity?

First, as we have pointed out, the soundness proofs for such assertions for the language constructs follow certain patterns. We extract these patterns and use in the design of the parametric proof rules and modularisation of proofs.

Secondly, we note that this assertion may be split into two. A pure numerical assertion should correspond exactly to the heap-space aware typing judgments from [4]. It will look similar to the one above, where in the relation $\Vdash$, defined for sets of variables and contexts, the disjoint union $\uplus$ is substituted with the union $\cup$. The second type system should approximate benign sharing as we have mentioned in the previous section. It may be linear typing or more general usage-aspect typing [12], etc.

We instantiate the parametric assertion with the usage-aspects-aware one. We consider 3 subsets of used variables. A set U_1 denotes variables that are potentially destroyed during the evaluation of a region, R_1. The region R_2 corresponding to U_2 is intact, but may be shared with the result region. The less "dangerous" use concerns variables in U_3: they refer to intact regions R_3 and do not share locations with the result. A set F denotes locations from a freelist, as in the example above.

Sets $U_1,\, U_2,\, U_3$, context Γ and type T play the role of parameters in the pre- and post-conditions. Recall, that the parametric specialised assertion has the form:

$$\mathit{Pre} \Rightarrow \mathit{Post} \equiv \lambda\, E\, h\, h'\, v. \forall X.\, (\mathit{Pre}\, E\, h) \longrightarrow (\mathit{Post}\, h'\, v).$$

We instantiate the parametric assertion in the following way:

$$\begin{array}{rcl}X & := & (R_1,\, R_2,\, R_3,\, F) \\ \mathit{Pre} & := & \mathit{Pre}_{UA} \ \ \Gamma\, U_1\, U_2\, U_3\, T \\ \mathit{Post} & := & \mathit{Post}_{UA} \ \ \Gamma\, U_1\, U_2\, U_3\, T\end{array}$$

where X_i is the i-th projection of X, $1 \leq i \leq 4$:

$$\begin{array}{l}\mathit{Pre}_{UA} \ \ \Gamma\, U_1\, U_2\, U_3\, T \equiv\ \lambda X. \\ \quad U_1 \cup U_2 \cup U_3 \subseteq \mathit{dom}\,\Gamma \wedge \\ \quad \mathit{freelist}(h,\, X_4) \wedge \\ \quad E,\, U_1 \Vdash_\Gamma^h X_1 \ \wedge\ E,\, U_2 \Vdash_\Gamma^h X_2,\, \mathsf{false} \ \wedge\ E,\, U_3 \Vdash_\Gamma^h X_3 \wedge \\ \quad X_1 \cap (X_1 \cup X_2) = \emptyset \ \wedge\ X_4 \cap (X_1 \cup X_2 \cup X_3) = \emptyset\end{array}$$

and

$$\begin{array}{lll}\mathit{Post}_{UA} \ \ \Gamma\, U_1\, U_2\, U_3\, T \equiv & & \lambda X. \\ \exists\, Q\, H. & & v \Vdash_T^{h'} Q,\, \mathsf{false} \wedge \\ & & \mathit{freelist}(h',\, H) \wedge \\ & & \mathit{footprint}(X_4 \cup X_1,\, h, h') \wedge \\ & & Q \subseteq (X_4 \cup X_1 \cup X_2) \wedge \\ & & H \subseteq (X_4 \cup X_1) \wedge \\ & & \mathit{dom}\, h = \mathit{dom}\, h' \wedge \\ & & E,\, U_2 \Vdash_\Gamma^{h'} X_2,\, \mathsf{true} \longrightarrow \\ & & \quad v \Vdash_T^{h'} Q,\, \mathsf{true}\end{array}$$

Here, the relation $\Vdash$ has the same meaning as above, except the last parameter. For the usage-aspect-2 variables it switches on/off the internal separation. For the the usage-aspect-1 and -3 it is irrelevant.

3.4 Parametric proof rules

We now discuss the design of soundness proofs for parametric rules. Consider the most intricate case – the rule for the `let`-

construct – in detail. We sketch the proof of the rule

$$\frac{\begin{array}{c} G \rhd e_1 : \mathit{Pre}_1 \Rightarrow \mathit{Post}_1 \\ G \rhd e_2 : \mathit{Pre}_2 \Rightarrow \mathit{Post}_2 \\ \mathsf{gLET}\ x\ \mathit{Pre}\ \mathit{Post}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2 \end{array}}{G \rhd \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \mathit{Pre} \Rightarrow \mathit{Post}}$$

in a backward style. The condition gLET is to be defined.

1. Apply the rule of consequence to obtain the subgoals

$$\frac{\begin{array}{c} G \rhd e_1 : \mathit{Pre}_1 \Rightarrow \mathit{Post}_1 \\ G \rhd e_2 : \mathit{Pre}_2 \Rightarrow \mathit{Post}_2 \\ \mathsf{gLET}\ x\ \mathit{Pre}\ \mathit{Post}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2 \end{array}}{G \rhd \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : ?P}$$

with a predicate $?P$ to be instantiated, and

$$\frac{\begin{array}{c} G \rhd e_1 : \mathit{Pre}_1 \Rightarrow \mathit{Post}_1 \\ G \rhd e_2 : \mathit{Pre}_2 \Rightarrow \mathit{Post}_2 \\ \mathsf{gLET}\ x\ \mathit{Pre}\ \mathit{Post}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2 \end{array}}{\forall\, E\, h\, h'\, v.\ ?P\ E\, h\, h'\, v \longrightarrow (\mathit{Pre} \Rightarrow \mathit{Post}\ E\, h\, h'\, v)}$$

2. The first subgoal is eliminated by the basic LET-rule, and $?P$ is instantiated with

$$\mathsf{LET}\ x\,(\mathit{Pre}_1 \Rightarrow \mathit{Post}_1)\,(\mathit{Pre}_2 \Rightarrow \mathit{Post}_2)$$

3. The subgoal to prove is

$$\frac{\mathsf{gLET}\ x\ \mathit{Pre}\ \mathit{Post}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2}{\begin{array}{l} \forall\, E\, h\, h'\, v. \\ \mathsf{LET}\ x\,(\mathit{Pre}_1 \Rightarrow \mathit{Post}_1)\,(\mathit{Pre}_2 \Rightarrow \mathit{Post}_2)\ E\, h\, h'\, v \\ \longrightarrow (\mathit{Pre} \Rightarrow \mathit{Post})\ E\, h\, h'\, v \end{array}}$$

4. Simple backward logical reasoning yields:

$$\frac{\begin{array}{l} \mathsf{LET}\ x\,(\mathit{Pre}_1 \Rightarrow \mathit{Post}_1)\,(\mathit{Pre}_2 \Rightarrow \mathit{Post}_2)\ E\, h\, h'\, v \\ \mathsf{gLET}\ x\ \mathit{Pre}\ \mathit{Post}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2 \end{array}}{(\mathit{Pre} \Rightarrow \mathit{Post})\ E\, h\, h'\, v}$$

5. Unfold the definition of LET [2]:

$$\frac{\begin{array}{l} \exists\, h_1\, w. \\ ((\mathit{Pre}_1 \Rightarrow \mathit{Post}_1)\ E\, h\, h_1\, w)\ \wedge \\ (w \neq \bot) \wedge \\ ((\mathit{Pre}_2 \Rightarrow \mathit{Post}_2)\ E\langle x := w\rangle\, h_1\, h'\, v) \end{array}}{(\mathit{Pre} \Rightarrow \mathit{Post})\ E\, h\, h'\, v}$$

6. Fix h_1, w and unfold the definition of $\Rightarrow$-operator. One obtains the following subgoal:

$$\frac{\begin{array}{ll} \forall\, X.\ (\mathit{Pre}_1\ X\ E\, h) \longrightarrow (\mathit{Post}_1\ X\, h_1\, w) & (1) \\ \forall\, X.\ (\mathit{Pre}_2\ X\ E\langle x := w\rangle\, h_1) \longrightarrow (\mathit{Post}_2\ X\, h'\, v) & (2) \end{array}}{\forall\, X.\ (\mathit{Pre}\ X\ E\, h) \longrightarrow (\mathit{Post}\ X\, h'\, v)}$$

7. Fix X and assume that $\mathit{Pre}\ X\ E\, h$ holds.
8. Show that $\mathit{Pre} X\ E\, h$ implies the existence of Y, s.t. $\mathit{Pre}_1 Y\ E\, h$ holds, that is, prove lemma_1 which is a part of gLET.
9. Eliminate the $\forall$-quantifier in (1) by instantiating the quantified parameter with Y.
10. Apply (1) to obtain $\mathit{Post}_1\ Y\, h_1\, w$.
11. Show that $\mathit{Pre}\ X\ E\, h$, $\mathit{Pre}_1\ Y\ E\, h$ and $\mathit{Post}_1\ Y\, h_1\, w$ imply the existence of Z, s.t. $\mathit{Pre}_2\ Z\, E\langle x := w\rangle\, h_1$, that is, prove lemma_2, which is another part of gLET.
12. Eliminate the $\forall$-quantifier in (2) by instantiating the quantified parameter with Z.
13. Apply (2) to obtain $\mathit{Post}_2\ Z\, h'\, v$.
14. Show that the conditions $\mathit{Pre} X\ E\, h$, $\mathit{Pre}_1 Y\ E\, h$, $\mathit{Post}_1 Y\, h_1\, w$ $\mathit{Pre}_2\ Z\, E\langle x := w\rangle\, h_1$ and $\mathit{Post}_2\ Z\, h'\, v$ imply $\mathit{Post}\ X\, h'\, v$, that is the last part of gLET, the lemma_3, holds.

[2] For the sake of convenience we will now omit gLET $x\ \mathit{Pre}\ \mathit{Post}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2$ in the assumptions of the subgoals below.

In the proof script above steps 1-7, 9, 10, 12, 13 do not depend on choice of Pre, Post, Pre_1, Post_1, Pre_2, Post_2. These steps form the proof of the parametric `let`-rule. The proof is sound if one is able to perform steps 8, 11, 14. This gives us a desirable soundness condition which is a conjunction of the corresponding three statements:

$$\begin{array}{l} \mathsf{gLET}\ x\ \mathit{Pre}\ \mathit{Post}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2\ \equiv \\ \quad \mathsf{lemma\_1}\ \mathit{Pre}\ \mathit{Pre}_1\ \wedge \\ \quad \mathsf{lemma\_2}\ x\ \mathit{Pre}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \wedge \\ \quad \mathsf{lemma\_3}\ x\ \mathit{Pre}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2\ \mathit{Post} \end{array}$$

with

$$\begin{array}{l} \mathsf{lemma\_1}\ \mathit{Pre}\ \mathit{Pre}_1 \equiv \\ \forall\, E\, h.\ \forall\, X.\ (\mathit{Pre}\ X\ E\, h) \longrightarrow \exists\, Y.\ (\mathit{Pre}_1\ Y\ E\, h) \end{array}$$

$$\begin{array}{lll} \multicolumn{3}{l}{\mathsf{lemma\_2}\ x\ \mathit{Pre}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2 \equiv} \\ \forall\, E\, h\, h_1\, w.\ \forall\, X\, Y. & (\mathit{Pre}\ X\ E\, h) \longrightarrow \\ & (\mathit{Pre}_1\ Y\ E\, h) \longrightarrow (\mathit{Post}_1\ Y\, h_1\, w) \longrightarrow \\ & \exists\, Z.\ (\mathit{Pre}_2\ Z\ E\langle x := w\rangle\, h_1) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{lemma\_3}\ x\ \mathit{Pre}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2\ \mathit{Post} \equiv} \\ \forall\, E\, h\, h_1\, w\, h'\, v. & \\ \forall\, X\, Y\, Z. & (\mathit{Pre}\ X\ E\, h) \longrightarrow \\ & (\mathit{Pre}_1\ Y\ E\, h) \longrightarrow (\mathit{Post}_1\ Y\, h_1\, w) \longrightarrow \\ & (\mathit{Pre}_2\ Z\ E\langle x := w\rangle\, h_1) \longrightarrow \\ & (\mathit{Post}_2\ Z\, h'\, v) \longrightarrow (\mathit{Post}\ X\, h'\, v) \end{array}$$

Similarly, for the `let`-rule for a non-pure type system:

$$\frac{\begin{array}{c} G \rhd e_1 : \mathit{Pre}_1 \Rightarrow \mathit{Post}_1 \\ G \rhd e_2 : \mathit{Pre}_2 \Rightarrow \mathit{Post}_2 \\ \mathsf{gLET}'\ x\ \mathit{Pre}\ \mathit{Post}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2 \mathit{Post}_2\ \mathcal{P} \end{array}}{G \rhd \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \mathit{Pre} \Rightarrow \mathit{Post}}$$

we introduce the soundness predicate gLET$'$, as conjunction of 3 lemmas, where the first lemma is the one for the pure type system, and

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{lemma\_2}'\ x\ \mathit{Pre}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathcal{P} \equiv} \\ \forall\, E\, h\, h_1\, w.\ \forall\, X\, Y. & (\mathit{Pre}\ X\ E\, h) \longrightarrow \\ & (\mathit{Pre}_1\ Y\ E\, h) \longrightarrow (\mathit{Post}_1\ Y\, h_1\, w) \longrightarrow \\ & (\mathcal{P}\ E\, h\, h_1\, w) \longrightarrow \\ & \exists\, Z.\ (\mathit{Pre}_2\ Z\ E\langle x := w\rangle\, h_1) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{lemma\_3}'\ x\ \mathit{Pre}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2\ \mathit{Post}\ \mathcal{P} \equiv} \\ \forall\, E\, h\, h_1\, w\, h'\, v. & \\ \forall\, X\, Y\, Z. & (\mathit{Pre}\ X\ E\, h) \longrightarrow \\ & (\mathit{Pre}_1\ Y\ E\, h) \longrightarrow (\mathit{Post}_1\ Y\, h_1\, w) \longrightarrow \\ & (\mathcal{P}\ E\, h\, h_1\, w) \longrightarrow \\ & (\mathit{Pre}_2\ Z\ E\langle x := w\rangle\, h_1) \longrightarrow \\ & (\mathit{Post}_2\ Z\, h'\, v) \longrightarrow (\mathit{Post}\ X\, h'\, v) \end{array}$$

The parametric rules for basic expressions are instances of the rule of consequence, applied to the appropriate rule in the basic logic.

We sum up the main result of this subsection. Recall, that $\mathit{PreSpec}(\alpha) = \alpha \to \mathit{PreSpec}$ and $\mathit{PostSpec}(\alpha) = \alpha \to \mathit{PostSpec}$. For the language constructs above we define the soundness predicates

$$\begin{array}{ll}
\texttt{gVAR}: & \mathit{Vars} \to \mathit{PreSpec}(\alpha) \to \mathit{PostSpec}(\alpha) \to \mathit{Bool} \\
\texttt{gPUTFI}: & \mathit{Vars} \to \mathit{Fields} \to \mathit{Vars} \to \\
 & \to \mathit{PreSpec}(\alpha) \to \mathit{PostSpec}(\alpha) \to \mathit{Bool} \\
\texttt{gIF}: & \mathit{Vars} \to \mathit{PreSpec}(\alpha) \to \mathit{PostSpec}(\alpha) \to \\
 & \to \mathit{PreSpec}(\alpha) \to \mathit{PostSpec}(\alpha) \to \\
 & \to \mathit{PreSpec}(\alpha) \to \mathit{PostSpec}(\alpha) \to \mathit{Bool} \\
\texttt{gLET}: & \mathit{Vars} \to \mathit{PreSpec}(\alpha) \to \mathit{PostSpec}(\alpha) \to \\
 & \to \mathit{PreSpec}(\alpha) \to \mathit{PostSpec}(\alpha) \to \\
 & \to \mathit{PreSpec}(\alpha) \to \mathit{PostSpec}(\alpha) \to \mathit{Bool}.
\end{array}$$

which assure the soundness of the corresponding parametric rules:

$$\frac{\mathsf{gVAR}\ x\ \ \mathit{Pre}\ \mathit{Post}}{G \rhd \mathtt{var}\ x:\ \mathit{Pre} \Rightarrow \mathit{Post}}\,\textsc{gVar}$$

$$\frac{\mathsf{gPUTFI}\ x\,t\,y\ \mathit{Pre}\ \mathit{Post}}{G \rhd x.t{:=}y:\ \mathit{Pre} \Rightarrow \mathit{Post}}\,\textsc{gPutFi}$$

$$\frac{\begin{array}{c}\mathsf{gIF}\ x\ \ \mathit{Pre}\mathit{Post}\ \mathit{Pre}_t\mathit{Post}_t\ \mathit{Pre}_f\mathit{Post}_f \\ G \rhd e_t:\ \mathit{Pre}_t \Rightarrow \mathit{Post}_t \qquad G \rhd e_f:\ \mathit{Pre}_f \Rightarrow \mathit{Post}_f\end{array}}{G \rhd \mathtt{if}\ x\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f:\ \mathit{Pre} \Rightarrow \mathit{Post}}\,\textsc{gIf}$$

$$\frac{\begin{array}{c}\mathsf{gLET}\ x\ \ \mathit{Pre}\ \mathit{Post}\ \mathit{Pre}_1\ \mathit{Post}_1\ \mathit{Pre}_2\ \mathit{Post}_2 \\ G \rhd e_1:\ \mathit{Pre}_1 \Rightarrow \mathit{Post}_1 \quad G \rhd e_2:\ \mathit{Pre}_2 \Rightarrow \mathit{Post}_2\end{array}}{G \rhd \mathtt{let}\ x = e_t\ \mathtt{in}\ e_f:\ \mathit{Pre} \Rightarrow \mathit{Post}}\,\textsc{gLet}$$

The rules for function call and method invocation are instances of appropriate basic rules. There are no special soundness predicates.

The predicates are designed as follows:

$$\begin{array}{l}
\mathsf{gVAR}\ x\ \ \mathit{Pre}\ \mathit{Post} \equiv\ \ \forall\, E\,h\,h\,v.\ \Big(\mathsf{VAR}\ x\ \ E\,h\,h'\,v\Big) \longrightarrow \\
\qquad\qquad\qquad\qquad \Big((\mathit{Pre} \Rightarrow \mathit{Post})\ E\,h\,h\,v\Big) \\
\mathsf{gPUTFI}\ x\ \mathit{Pre}\ \mathit{Post}\ t\ y \equiv \\
\qquad \forall\, E\,h\,h\,v.\ \Big(\mathsf{PUTFI}\ x\,t\,y\,E\,h\,h'\,v\Big) \longrightarrow \\
\qquad \Big((\mathit{Pre} \Rightarrow \mathit{Post})\ E\,h\,h\,v\Big) \\
\mathsf{gIF}\ x\ \ \mathit{Pre}\mathit{Post}\ \mathit{Pre}_t\mathit{Post}_t\ \mathit{Pre}_f\mathit{Post}_f \equiv \\
\qquad \forall\, E\,h\,h'\,v. \\
\qquad \Big(E\langle x\rangle = \mathsf{true} \longrightarrow \\
\qquad \big((\mathit{Pre}\,E\,h\,h\,v) \to (\mathit{Pre}_t\,E\,h\,h\,v)\big)\wedge \\
\qquad \big((\mathit{Post}_t\,E\,h\,h\,v) \to (\mathit{Post}\,E\,h\,h\,v)\big)\Big)\wedge \\
\qquad \Big(E\langle x\rangle = \mathsf{false} \longrightarrow \\
\qquad \big((\mathit{Pre}\,E\,h\,h\,v) \to (\mathit{Pre}_f\,E\,h\,h\,v)\big)\wedge \\
\qquad \big((\mathit{Post}_f\,E\,h\,h\,v) \to (\mathit{Post}\,E\,h\,h\,v)\big)\Big)\wedge
\end{array}$$

We have used this parametric framework to obtain the soundness proof of the usage-aspect-aware logic of [12].

3.5 Combined assertions

We start this subsection with a perhaps surprising remark: the soundness of the combined `let`-rule in general does not follow from the soundness of the rules for its components. For the sake of simplicity one may think that the first `let` rule does not contain a semantical condition. To provide some intuition, we consider the first-order skeleton of the problem, that is, we should show that from

$$\frac{\begin{array}{c}A_1^i \longrightarrow B_1^i \\ A_2^i \longrightarrow B_2^i\end{array}}{A^i \longrightarrow B^i} \quad S'i$$

with $i = 1,\ 2$, it follows

$$\frac{\begin{array}{c}A_1^1 \wedge A_1^2 \longrightarrow B_1^1 \wedge B_1^2 \\ A_2^1 \wedge A_2^2 \longrightarrow B_2^1 \wedge B_2^2\end{array}}{A^1 \wedge A^2\ \longrightarrow\ B^1 \wedge B^2}\,\mathrm{S'C}$$

The assumption $A_1^1 \wedge A_1^2\ \longrightarrow\ B_1^1 \wedge B_1^2$ does not imply the assumptions of $S'1$ and $S'2$. This makes these rules useless and the conclusion of the combined implication unprovable. Similarly, $G \times G'\ \rhd\ e_1\ :\ \mathit{Pre}_1^1 \cdot \mathit{Pre}_1^2 \Rightarrow \mathit{Post}_1^1 \cdot\cdot \mathit{Post}_1^2$ does not imply $G\ \rhd\ e_1\ :\ \mathit{Pre}_1^1 \Rightarrow \mathit{Post}_1^1$. Therefore, the `let`-rules for the components does not justify the `let`-rule for the combination.

A similar problem takes place for the `if`-rule, while for the rules of the basic language constructs the soundness of the components do imply the soundness of the composition.

Therefore, if one wants to re-use the soundness of the components, one must find a sufficient condition of soundness which is stronger than one needs for a single rule.

The soundness condition that we have defined above, has exactly the property we need: the soundness condition of the combined assertion follows from the soundness conditions of its components. Below we give a table with the corresponding formal statements. We have included examples with the basic expressions as well, to keep the things uniform.

$$\frac{\mathsf{gVAR}\ x\ \ \mathit{Pre}^1\ \mathit{Post}^1 \quad \mathsf{gVAR}\ x\ \ \mathit{Pre}^2\ \mathit{Post}^2}{\mathsf{gVAR}\ x\ \ (\mathit{Pre}^1 \cdot \mathit{Pre}^2)\ (\mathit{Post}^1 \cdot\cdot \mathit{Post}^2)}\,\textsc{cVar}$$

$$\frac{\mathsf{gPUTFI}\ x\,t\,y\ \mathit{Pre}^1\ \mathit{Post}^1 \quad \mathsf{gPUTFI}\ x\,t\,y\ \mathit{Pre}^2\ \mathit{Post}^2}{\mathsf{gPUTFI}\ x\,t\,y\ (\mathit{Pre}^1 \cdot \mathit{Pre}^2)\ (\mathit{Post}^1 \cdot\cdot \mathit{Post}^2)}\,\textsc{cPutFi}$$

$$\frac{\begin{array}{c}\mathsf{gIF}\ x\ \ \mathit{Pre}^1\mathit{Post}^1\ \mathit{Pre}_t^1\mathit{Post}_t^1\ \mathit{Pre}_f^1\mathit{Post}_f^1 \\ \mathsf{gIF}\ x\ \ \mathit{Pre}^2\mathit{Post}^2\ \mathit{Pre}_f^2\mathit{Post}_f^2\ \mathit{Pre}_f^2\mathit{Post}_f^2\end{array}}{\begin{array}{ll}\mathsf{gIF}\ x & (\mathit{Pre}^1 \cdot \mathit{Pre}^2)(\mathit{Post}^1 \cdot\cdot \mathit{Post}^2) \\ & (\mathit{Pre}_t^1 \cdot \mathit{Pre}_t^2)(\mathit{Post}_t^1 \cdot\cdot \mathit{Post}_t^2) \\ & (\mathit{Pre}_f^1 \cdot \mathit{Pre}_f^2)(\mathit{Post}_f^1 \cdot\cdot \mathit{Post}_f^2)\end{array}}\,\textsc{cIf}$$

$$\frac{\begin{array}{c}\mathsf{gLET}\ '\ x\ \mathit{Pre}^1\ \mathit{Post}^1\ \mathit{Pre}_1^1\ \mathit{Post}_1^1\ \mathit{Pre}_2^1\mathit{Post}_2^1\ \mathcal{P} \\ \mathsf{gLET}\ x\ \mathit{Pre}^2\ \mathit{Post}^2\ \mathit{Pre}_1^2\ \mathit{Post}_1^2\ \mathit{Pre}_2^2\ \mathit{Post}_2^2 \\ \mathit{Pre}^2\ \mathit{Pre}_1^2\ \mathit{Post}_1^2\ \mathsf{implies}\ \mathcal{P}\end{array}}{\begin{array}{ll}\mathsf{gLET}\ x & (\mathit{Pre}^1 \cdot \mathit{Pre}^2)(\mathit{Post}^1 \cdot\cdot \mathit{Post}^2) \\ & (\mathit{Pre}_1^1 \cdot \mathit{Pre}_1^2)(\mathit{Post}_1^1 \cdot\cdot \mathit{Post}_1^2) \\ & (\mathit{Pre}_2^1 \cdot \mathit{Pre}_2^2)(\mathit{Post}_2^1 \cdot\cdot \mathit{Post}_2^2)\end{array}}\,\textsc{cLET}$$

where the approximation

$$\begin{array}{ll}
\mathit{Pre}^2\ \mathit{Pre}_1^2\ \mathit{Post}_1^2\ \mathsf{implies}\ \mathcal{P} \equiv & \forall\, E\,h\,h_1\,w.\ \forall\, X\,Y. \\
 & (\mathit{Pre}^2\ X\,E\,h) \to \\
 & (\mathit{Pre}_1^2\ Y\,E\,h) \to (\mathit{Post}_1^2\ Y\,h_1\,w) \\
 & \to (\mathcal{P}\ E\,h\,h_1\,w)
\end{array}$$

of the semantical side condition allows us to eliminate the side condition in the combined rule.

4. Related Works

The current work is motivated by the experience we have collected in *Mobile Resource Guarantees* (MRG) project [1], when working on heap-space predicates. In a recent work [3] we have proposed a synthesis between type systems and program logics. The heap-space resource type system from [4] is used for automated generation of invariants and proof scripts in the logic for `Grail`. The idea of mapping type systems onto program logics in general case is sketched in [5]. In [6] the authors consider combinators for general form of assertions and rules. In this paper we develop these ideas and consider combinators for specialised logics.

In [9] the authors define a safety logic for an assembly language with a parametric verification condition generator. The VCG is constructed in a "traditional way" with small-step weakest precondition generation. The safety predicates to be checked are of general form.

Static analysis and its generalisation in the form of Relational Hoare Logic is used in [10] for checking correctness of optimising transformations. The transformations are presented by rules for deriving (non-standard) typed equations. There the logic is considered as a generalisation of typing systems, while we use logics to express the semantics of type judgments or to turn semantical properties into typeable ones.

5. Conclusions and Future Work

We have considered a parametric logical framework for checking soundness of type systems. It assists in the design of specialised logics that mirror types systems in a higher-order logic and may be used for automated program verification.

The framework helps to treat compound inference systems, where one of the components is a non-pure type system, but contains semantical conditions in assumptions of its rules. A well-designed composition may eliminate semantical conditions.

The parametric logic is implemented as an Isabelle theory. We have been testing the framework by applying it to different type systems. The first results obtained by instantiation the parametric assertion and rules with the usage-aspects aware typing system of [12], which generalises linear typing, are very encouraging.

We plan to design the specialised logic for the non-pure resource-aware type system of [4] and combine it with usage aspects. We want to combine this inference system with other type systems approximating benign sharing.

We are also interested in applying the framework to other type systems and their combinations.

6. Acknowledgements

The author is supported by the Mobile Resource Guarantees project (MRG, project IST-2001-33149), which is funded by the EU under the Global Computing pro-active initiative on the Future and Emerging Technologies.

The author would like to thank Martin Hofmann and Alberto Momigliano for a few fruitful discussions.

References

[1] Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., and Stark., I. Mobile Resource Guarantees for Smart Devices. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, number 3362 in LNCS, pages 1-26. Springer-Verlag, 2004.

[2] Beringer, L., Hofmann, M., Loidl, H.-W., and Momigliano, A. A Program Logic for Resource Verification. In *Proceedings of 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004)*, pages 34-49. Springer-Verlag LNCS, September 2004.

[3] Beringer, L., Hofmann, M., Momigliano, A., and Shkaravska, O. Automatic Certification of Heap Consumption. In *Logic for Programming, Artificial Intelligence and Reasoning: 11th International Conference, LPAR 2004*, volume 3452, pages 347-362. Springer-Verlag, 2005.

[4] Hofmann, M., and Jost, S. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, volume 38, pages 185-197. ACM Press, 2003.

[5] Hofmann, M. What do program logic and type systems have in common? In *Proceedings of ICALP'04*, pages 4-7, 2004.

[6] Beringer, L., and Momigliano, A. Implement a theorem prover. In *Mobile Resource Guarantee Project Deliverable D2e*, November 2003.

[7] Loidl, H.-W., and MacKenzie, K. A Gentle Introduction to Camelot. September, 2004. http://groups.inf.ed.ac.uk/mrg/camelot/Gentle-Camelot/camelot-gentle-intro.html.

[8] MacKenzie, K., and Wolverson, N. Camelot and Grail: resource-aware functional programming on the JVM. In *Trends in Functional Programming*, volume 4, pages 29-46. Intellect, 2004.

[9] Wildmoser, M., and Nipkow, T. Certifying Machine Code Safety: Shallow versus Deep Embedding. In K. Slind and A. Bunker and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004*, volume 3223 of LNCS, pages 305-320. Springer, 2004.

[10] Benton, N. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*. ACM, January, 2004.

[11] Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.-W., and Momigliano, A. A Program Logic for Resource. Submitted, 2005.

[12] Aspinall, D., and Hofmann, M. Another Type System for In-Place Update. In *ESOP'02 — European Symposium on Programming*, LNCS 2305, pages 36-52, Springer, 2002.

[13] Xi, H., and Pfenning, F. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–277, San Antonio, January, 1999.