IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D17 (WP6): Assertion Language

Due date of deliverable: March 2007
Actual submission date: October 2007

Start date of project: 1st March 2005 Duration: 48 months

Lead contractor: Ludwig-Maximilians Universität, München Revision: 1.18

Purpose: In order to develop proper certification mechanisms, we must define notations that allow resource consumption in the Hume Abstract Machine (HAM), and assembler programs translated from the HAM, to be expressed formally. These notations must use the cost model developed in WP2, and be sufficiently precise to allow direct use in an automatic theorem proving environment as a basis for checking resource certificates.

Results: This document describes assertion languages for use with Hume. Matching the two-level language structure of Hume, we define two assertion languages: one for the expression- and one for the coordination-layer. Both these notations are formalised within the Isabelle/HOL theorem prover. Examples are given to show Hume resource consumption constraints can be expressed using these notations.

Conclusion: The assertion notations are sufficiently powerful to allow the description of resource consumption for execution time, heap space and stack space. They are already formalised in a theorem prover and thus directly usable in a certification infra-structure.

Project co-funded by the European Commission within the 6^{th} Framework Programme (2002-06)			
Dissemination Level			
PU	Public		⁎
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Assertion Language

Hans-Wolfgang Loidl <hwloidl@tcs.ifi.lmu.de>,
Institut für Informatik, Theoretische Informatik, Ludwig-Maximilians Universität, München

Gudmund Grov <gudmund@macs.hw.ac.uk>,
School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland

Abstract

We define assertion notations that are capable of expressing resource consumption in Hume, as the basis for developing an infrastructure for certifying bounded resource consumption of Hume code. In order to match the two-level structure of Hume, we define two distinct notations that allow assertions to be attached to either Hume expressions or to Hume boxes, as appropriate. Expression-level constructs describe evaluation and resource consumption within a box, in a similar way to a strict, higher-order functional programming language, such as ML. We therefore use techniques taken from the semantics of and programming logics for such languages in defining our assertion notation. In particular, we develop a VDM-style program logic for Hume expressions, and show how to express resource consumption using a shallow embedding into the Isabelle/HOL theorem prover. Hume coordination-level constructs describe interactions between boxes, which form a cooperating network of processes. In order to describe coordination-level behaviour, we use Lamport's Temporal Logic of Actions (TLA) as an assertion notation. We show how to express liveness and safety properties of a network of boxes in this notation. The role of this deliverable in the overall workplan is discussed in the summary in Section 5.

Major Revisions		
Revision	Date	Changes
1.17	14 Aug. 2009	table-of-contents, positioning in Sec 5.2 (addressing Review Report Year 4)
1.15	30 July 2008	relationship to D26 in Sec 5.1 (addressing Review Report Year 3)
1.13	31 Oct. 2007	initial version

Contents

1 Introduction

The goal of WP6 is to devise a process for *certifying* resource bounds on Hume programs. In order to achieve this goal, we adopt a proof-carrying-code (PCC) approach, whereby a certificate is, in essence, a compressed version of a formal proof of the required bounds on resource consumption. Certificates in this form can easily be attached to library code, so providing a compositional mechanism for ensuring that constraints on resource consumption of external code can be met by the overall system.

Our approach exploits state-of-the-art technologies in automated theorem proving. We have therefore developed our formalisms using the well-known Isabelle/HOL automatic theorem-prover from the outset. One immediate advantage of this approach is that we can devise the assertion language as a *shallow embedding* into the higher-order logic (HOL) of the prover. A consequence is that we can use arbitrary HOL constructs to state the properties of a given expression. This approach gives great flexibility in the assertion language. It also avoids the cumbersome task of defining an assertion language as an explicit data structure and then providing an interpretation of this structure for both Hume levels. Using a shallow embedding, the key steps in defining an assertion language are i) to define the type of an assertion within HOL; ii) to define the format of a judgement that states that an assertion is valid for a program expression; and iii) finally, to devise a program logic that permits arbitrary assertions to be proved.

One important deviation from the original workplan is that we use a simplified intermediate code form, Core-HUME (defined in D11 [JLH07a]), rather than the Hume Abstract Machine (HAM), as the source notation for describing resource consumption. While not affecting the cost information that we can obtain, working at a higher level allows us to exploit structural information that is lost in the compilation to an abstract machine. We can then exploit standard program verification techniques when designing and using our assertion language. In order to ensure that the cost information provided in Core-HUME precisely matches the costs of the abstract machine, we build our formal model directly on the cost model developed for Core-HUME in D11 (Resource Analysis).

While the existence of an underlying theorem prover aids us in developing the assertion language as a shallow embedding, certain aspects of the formalisation of an entire programming language are cumbersome. We therefore impose some structural restrictions on the source language for which assertions should be produced. These restrictions are mainly met already by the transformations performed in the Hume compiler, but it has also proved necessary to place additional internal compiler annotations on some intermediate language constructs (Deliverable D11 describes the translation from Hume to Core-HUME in more detail [JLH07a]). The language formalised in this document is a subset of Core-HUME without such compiler annotations. We will discuss the restrictions in more detail in Section 3.

The structure of this document is as follows. Section 2 provides an overview of proof-carrying code approaches, and relates our approach to other existing approaches. Section 3 develops an assertion language for Hume expressions and gives examples on how to phrase resource bounds in this assertion language. Section 4 similarly provides an assertion language for Hume boxes. Finally, Section 5 concludes.

2 A Proof-carrying-code architecture

2.1 Background

In producing certificates for the bounded resource consumption of Hume programs we take a proof-carrying-code (PCC) approach. In this approach, a certificate is a condensed form of a formal proof for a certain safety policy. In our case, the safety policy is bounded resource consumption. In the following section, we will discuss the details of how we define this safety policy and show how statements in the policy appear. Before that, we give an overview of our approach to proof-carrying code, which

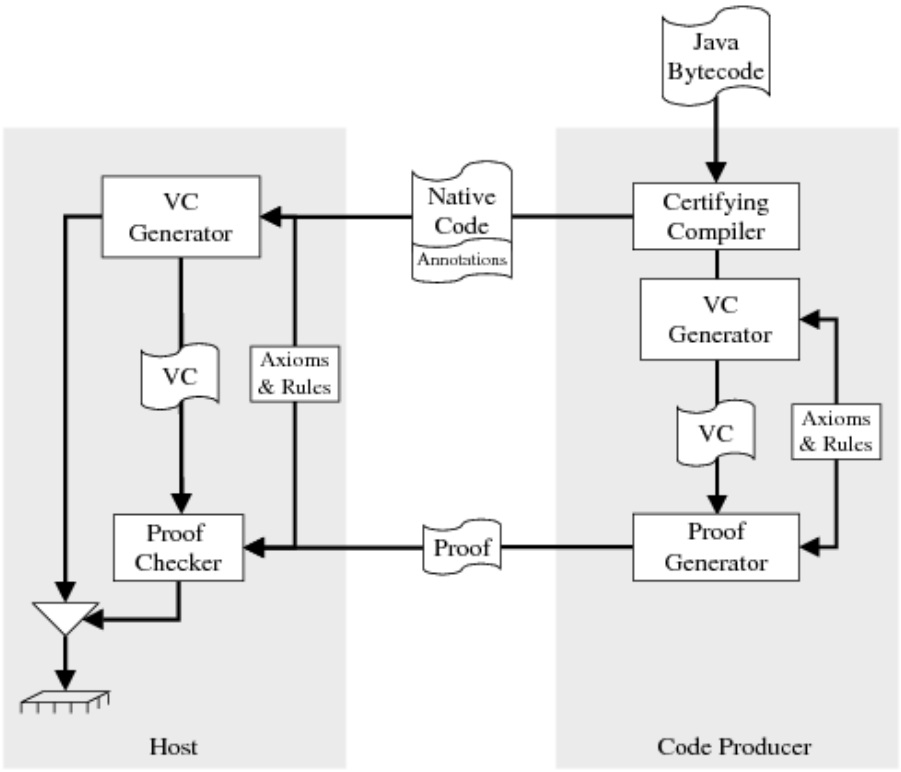

Figure 1: A classical proof-carrying code architecture

characterises our work in WP6.

The *classical* approach to proof-carrying code, as pioneered by Necula [Nec97], states that a safety policy shall be defined as a logic on the programming language that is used to transmit code. Typically low-level features are studied, such as the well-formedness of the heap. Typically the language is low-level, too, such as JVM bytecode. The rules of the logic are typically structural over the programming language, and state that the safety policy is maintained for all rules. In order to ensure this, most rules will need to impose side conditions on the code. For example, if the safety police is well-formedness of the heap, then the code for constructing a new list-cell must ensure that the result value is a list cell, with the value in the first component and a well-formed list structure in the second component. Using standard automated theorem proving machinery, a verification condition generator (VCG) will apply the (structural) rules of the logic and collect all these side conditions. This generated set of verification conditions is what the proof checker has to prove, by applying the information that is present in the transmitted certificate. There are several choices for the level of detail provided in the certificate, and we will discuss these below. Figure 1 (originally from [Nec97]) gives an overview of such an architecture.

This approach has several weaknesses. First, the trusted code base (TCB), ie. the software whose correctness the user must rely on, is fairly large: it consists of the verification condition generator and the proof checker. Although neither is not conceptually difficult, they each represent a large piece of software, and there is a danger that due to a bug in the verification condition generator, the host will allow the imported code to be executed despite it being provably unsafe. Secondly, the design of the logic specifying the safety policy is non-trivial. Therefore, a formal soundness proof is needed in order to rely on the logic as basis for the PCC architecture.

These observations have initiated research in the the direction of *foundational PCC* [App01]. The overriding design goal here is to minimise the trusted code base and therefore improve the trust in

the PCC architecture overall. In this approach, the safety policy is defined directly on the operational semantics of the language. Thus, there is no need for an explicit soundness proof, since the "rules" in this logic are derived theorems. The cost of this foundational approach is that basic theorem proving machinery is absent in this framework. Abstractions that are used to define a program logic eg. as a Hoare-style logic [Hoa69], cannot be used and the proof therefore becomes more verbose. Even if it is possible to generate the proof automatically, this is a problem because it increases the size of the certificate.

The choice of the certificate format is crucial for the size of the generated certificates. In principle, terms and types of logical frameworks [Pfe01] can be used to encode properties and their proofs. This is conceptually appealing because it provides a generic framework for formalising proofs for an arbitrary logic, and a generic proof checker can be used. In fact, in such a logical framework the task of checking a proof amounts to checking the type of a given LF-Term. However, even simple proofs encoded in this generic format tend to become huge, several orders of magnitude larger than the program to be transmitted. Therefore, alternatives for representing certificates have been studied. In particular, oracle strings [Nec05] can significantly reduce proof size. The idea is, that instead of encoding the entire proof, it suffices the encode the decision points a proof strategy has while traversing a program. The proof checker then takes this information, encoded in an oracle string, and applies it at the relevant points.

2.2 A multi-layered logics approach to PCC

Our *multi-layered logics approach* [SHA+05, HLB05] to PCC combines advantages of both classical and foundational PCC, as discussed in the previous section. Most notably, we achive high confidence in the infrastructure by encoding the logics in an automated theorem prover (Isabelle/HOL). We achieve potentially short certificates, by tightly integrating the process of analysing resource bounds, with the process of finding a proof for a given resource bound. First we discuss the principles of our approach.

We use a *multi-layered logics approach* (shown in Figure 2), where all logics are formalised in a proof assistant, and meta-theoretic results such as soundness provide the desired confidence. We developed this approach in the context of a predecessor project (the EU Framework V FET-Open Mobile Resource Guarantees project [SHA+05]). Its most notable feature is the fact that the logic to be used for proving resource bounds, is tailored to the high-level structure that is used to infer these bounds in the first place. The examples in Section 3.5 will show this correspondence in more detail. Providing an explicit logic gives us access to the abstractions used in classical PCC to define the safety policy. However, since we embed our logics into a prover, and provide soundness results, we also gain confidence in the correctness of the result. We do not need an explicit verification condition generator, but rather are able to prove the properties on higher level. This abstraction is justified, since each step on the higher level has been proven sound, and in principle it would be possible to send the entire soundness proof together with the certificate, proper.

As the basis we have the *operational semantics* which is extended with resources to capture the aspect of resource consumption in a program. Judgements in the operational semantics have the form $E, h \vdash \mathsf{e} \Downarrow (v, h', \rho)$, where E maps variables to values, h represents the pre-heap and h' the post-heap, and v is the result value, consuming ρ resources. We discuss the operational semantics for the Core-HUME language in Section 3.2. The foundational PCC approach [App01] performs proofs directly on this level thereby reducing the size of the trusted code base, but thereby increasing the size of the generated proofs considerably. To remedy this situation more recent designs, such as the Open Verifier Framework [CCNS05] or Certified Abstract Interpretation [BJP06], add untrusted, but provably sound, components to a foundational PCC design.

On the next level there is a general-purpose *program logic* for partial correctness. Judgements in this logic have the form $\Gamma \ \rhd \ \mathsf{e} : A$, where the context Γ maps expressions to assertions, and A, an assertion, is a predicate over the parameters of the operational semantics. We discuss the program logic

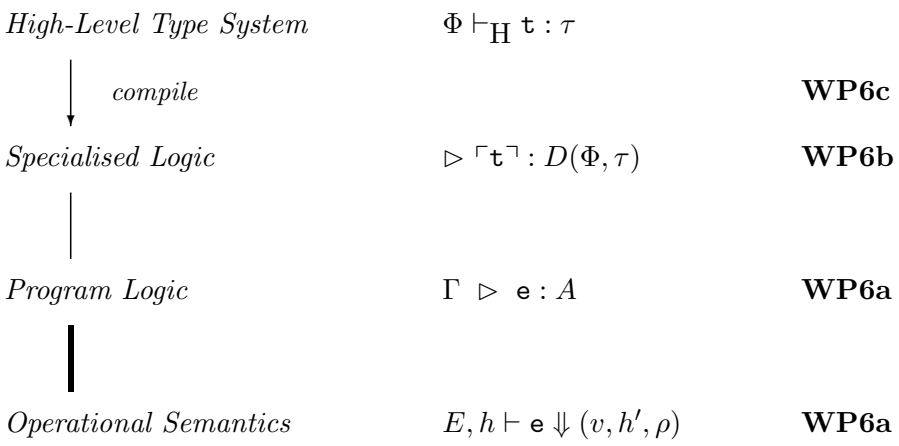

Figure 2: A family of logics for resource consumption

for the Core-HUME language in Section 3.3. The role of the program logic is to serve as a platform on which various higher level logics may be unified. An important facet in the safe design of such a multi-layered-logics approach is a soundness proof of this general program logic, as the trustworthiness of any application logic defined at higher levels depends upon it. Note that, since we formalise the entire hierarchy of logics and prove soundness, we do not need to include any of these logics in the trusted code base.

On top of the general-purpose logic, we define a *specialised logic*, for example a logic that captures the heap consumption of an expression. This logic uses a restricted format of assertions, called *derived assertions*, which reflects the judgement of the high-level type system. Judgements in the specialised logic have the form $\triangleright \ulcorner \mathtt{t} \urcorner : D(\Phi, \tau)$, where the expression $\ulcorner \mathtt{t} \urcorner$ is the result of compiling a high-level term `t` down to a low-level language, and the information in the high-level type system is encoded in a special form of assertion $D(\Phi, \tau)$ that relies on the context Φ and type τ associated to `t`. 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 complete, but it should provide support for automated proof search. In the case of the logic for heap consumption, we can achieve this by inferring a system of derived 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-directedness of the typing rules. At points where syntax-directedness fails — such as recursive program structures — the necessary invariants are provided by the type system.

On the top level we find a *high-level type system* that encodes information on resource consumption. In the judgement $\Phi \vdash_H \mathtt{t} : \tau$, the term `t` has an (extended) type τ in a context Φ. The type system for Core-HUME describing heap space consumption, as discussed in D11 [JLH07a], is an example for such a high-level type system. This is an example of increasingly complex type systems that have found their way into main-stream programming as a partial answer to the unfeasibility of proving general program correctness. Given this complexity, soundness proofs of the type systems become subtle. As we have seen, our approach towards guaranteeing the absence of bad behaviour at the compiled code level is to translate types into proofs in a suitably specialised program logic.

3 Formalisation and Assertion Language for Hume Expressions

In this section we present a formalisation of *Core-HUME*, the compiler intermediate language that is used for resource analysis of Hume programs. By using the same level of abstraction as in the analysis, it is possible to directly exploit the underlying cost model. This formal correspondence also provides a promising basis for the generation of the certificates that will be required later in WP6: a trace of the analysis can be used as a resource bound certificate, since it encodes the decisions that need to be taken in order to find a resource bound.

3.1 Source language: Core-HUME

The source language used in this formalisation is Core-HUME. For the purposes of this formalisation Core-HUME has been stripped of those annotations that are used only internally by the Hume compiler. This does not affect the cost model or analysis in any way. Core-HUME is an intermediate language that is used as part of the translation of Hume to Hume Abstract Machine (HAM) code. While Core-HUME is closer to the HAM than source Hume code, it retains more of the high-level program structure than in compiled HAM code. For example, in Core-HUME, the structure of a conditional expression is recorded with two branches, rather than, as in the HAM, by translation into a sequence of HAM instructions linked by (conditional) branch instructions. The same approach has already been adopted for the analysis of source Hume programs in WP3 and WP4. It follows that we can directly mirror the assignment of costs to language constructs that has been developed and validated for these analyses.

$$\text{Patt} \ni p ::= x \mid v \mid c\ x_1 \ldots x_n \mid \_$$

$$\begin{aligned}\text{Expr} \ni e ::= {} & x \mid v \mid c\ x_1 \ldots x_n \mid x\ x_1 \ldots x_n \mid x_1 \oplus x_2 \mid \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\ & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{case}\ x\ \texttt{of}\ p_1 \to e_1\ \texttt{otherwise}\ e_2\end{aligned}$$

Discussion: A pattern p is either a variable $x \in \text{Var}$, a value $v \in \text{Val}$, a constructor application $c\ x_1 \ldots x_n$ or a wildcard _. An expression Expr is either a variable x, a value v, a constructor application $c\ x_1 \ldots x_n$, a (higher-order) function call of x with arguments $x_1 \ldots x_n$, a binary primitive operation $\oplus$, a conditional, a let-expression, or a case expression. The latter is a one-step matching expression, with a default branch to be used if the match was unsuccessful. The entire program is represented by a table funTab, which maps a function name to a pair of a list of variable names, the formal parameters, and an expression, the function body.

3.2 Operational Semantics

In this section we give a big-step operational semantics of Core-HUME. First we define basic data structures used in the operational semantics.

$$\begin{aligned} l \in \text{Locn} &= \mathbb{N} \uplus \{\text{nil}\} \\ h, h', hh \in \text{Heap} &= \text{Locn} \rightsquigarrow_f \text{Val} \\ E, E' \in \text{Env} &= \text{Var} \Rightarrow \text{Val} \\ p \in \text{Resources} &= (\mathit{clock} :: \mathbb{Z},\ \mathit{callc} :: \mathbb{Z}, \mathit{maxstack} :: \mathbb{Z}) \\ v \in \text{Val} &= \{\bot\} \uplus \mathbb{Z} \uplus \mathbb{B} \uplus \text{Locn} \uplus (\text{Constr},\ \text{Locn}^*) \uplus (\text{Funs},\ \mathbb{N},\ \text{Locn}^*) \end{aligned}$$

Semantic Domains: We use disjoint sets of identifiers for variable names ($x \in \mathrm{Var}$), function names ($f \in \mathrm{Funs}$), and data constructor names ($c \in \mathrm{Constr}$). A location $l \in \mathrm{Locn}$ is either a natural number (representing an address in the dynamic heap), or the constant nil. A heap $h \in \mathrm{Heap}$ is a finite mapping (denoted by infix $\rightsquigarrow_f$) of locations to values. An environment $E \in \mathrm{Env}$ is a total function (denoted by infix $\Rightarrow$) of variable names to values. A semantic value ($v \in \mathrm{Val}$) is either undefined ($\bot$), an integer value ($\mathbb{Z}$) a boolean value ($\mathbb{B}$) with elements true and false, a location (Locn), a constructor $c \in \mathrm{Constr}$ applied to arguments at locations $l_1 \in \mathrm{Locn} \ldots l_n \in \mathrm{Locn}$, or a *closure*, a function value that has not yet been provided with all its arguments. Closures are represented as a tuple of function name, $f \in \mathrm{Funs}$, the number of fixed arguments $i \in \mathbb{N}$, and the values of these fixed arguments at locations $l_1 \in \mathrm{Locn} \ldots l_n \in \mathrm{Locn}$. We use the notation Locn^* for the domain of sequences of elements in Locn. A resource vector $p \in \mathrm{Resources}$ is a record with fields modelling the consumption of a specific resource: *clock* for time, *callc* for the number of function calls, and *maxstack* for the maximal stack usage. Note that we model the current heap usage of h as the size of its domain, ie. $|\mathrm{dom}\ h|$. The operation freshloc has the property freshloc $s \notin s$, for all s. The operations fst and snd extract the first and second element out of a pair, respectively. For combining two resource vectors we use the binary operation $\smile$, which computes the sum over the *clock* and *callc* fields, and the maximum over the *maxstack* field.

Notation: In function calls we use lists in the meta-language to represent argument lists. We write [] for the empty list, infix : for list-cons, infix ++ for list-concatenation, prefix # for the length of a list, and $f \star xs$ for list-map of the function f over the list xs. We use $\emptyset$ to denote the empty environment, and $E(x := r)$ to denote the update of x with r in the total map E. We use $h\ l$ for a lookup of l in the finite map h (the result Some v represents a proper value v, and None represents a failed lookup), and $h(l \mapsto v)$ to denote the update of l with v in the finite map h. We abbreviate sequences such as $xs_1 \ldots xs_n$ as $\vec{xs}$. For the binding of the values of function arguments $\vec{rs}$ to their formal parameters $\vec{xs}$ in an environment E yielding environment E' we extend the function update notation to sequences, i.e. we write $E(\vec{xs} := \vec{rs})$ for a a simultaneous update of a list of variables $\vec{xs}$ by a list of values $\vec{rs}$.

The following set of rules defines the semantics of pattern matching. A judgement of the form $\texttt{MATCH}\ E, h \vdash p\ \texttt{at}\ l \Downarrow (E', v)$ reads as follows: given a (variable) environment E and a heap h, the pattern-match of the value at location l in the heap h against the pattern p evaluates to the value v and modifies the environment to E'. The value result is only used to distinguish successful from unsuccessful pattern-matching. The result environment E' is used in the body of the branch.

$$\frac{h\ l = \mathrm{Some}\ v}{\texttt{MATCH}\ E, h \vdash v\ \texttt{at}\ l \Downarrow (E, l)} \quad (\textsc{pval})$$

$$\frac{h\ l \neq \mathrm{Some}\ v}{\texttt{MATCH}\ E, h \vdash v\ \texttt{at}\ l \Downarrow (E, \bot)} \quad (\textsc{pvalfail})$$

$$\frac{}{\texttt{MATCH}\ E, h \vdash x\ \texttt{at}\ l \Downarrow (E(x := l), l)} \quad (\textsc{pvar})$$

$$\frac{}{\texttt{MATCH}\ E, h \vdash \_\ \texttt{at}\ l \Downarrow (E, l)} \quad (\textsc{pwild})$$

$$\frac{\exists \vec{rs}.\ h\ l = \mathrm{Some}\ (c,\ \vec{rs})\ \wedge\ E' = E(\vec{vs} := \vec{rs})}{\texttt{MATCH}\ E, h \vdash\ c\ \vec{vs}\ \texttt{at}\ l \Downarrow (E', l)} \quad (\textsc{pcon})$$

$$\frac{\neg(\exists \vec{rs}.\ h\ l = \mathrm{Some}\ (c,\ \vec{rs}))}{\texttt{MATCH}\ E, h \vdash\ c\ \vec{vs}\ \texttt{at}\ l \Downarrow (E, \bot)} \quad (\textsc{pconfail})$$

The following set of rules defines the operational semantics for expressions. A judgement of the form $E, h \vdash e \Downarrow_m (v, h', p)$ can be read as follows: given a variable environment, E, and a heap, h, e evaluates in m steps to the value v, yielding the modified heap, h', and consumes resources p. The index m is only used for technical reasons (as a basis for induction in the soundness proof).

For capturing the resource consumption of Core-HUME instructions we use a table $\mathcal{R}$. In order to model costs for different control flows through an expression, we use a separate "resource constant" data-type, which is an enumeration mainly consisting of the expression formers, but extended to distinguish for example the true from the false case in a conditional. For example, $\mathcal{R}^{\texttt{if true}}$ covers the costs of a conditional in the case where the header evaluates to true, whereas $\mathcal{R}^{\texttt{if false}}$ covers the costs of a conditional in the case where the header evaluates to false, accounting for a conditional jump to the else branch that is performed on HAM level. This table ($\mathcal{R}$) therefore corresponds directly to the cost table for the HAM (also called $\mathcal{R}$) that was specified in Deliverable D4 [LH06]. We construct the table for primitive Core-HUME expressions by composing the individual costs for all the HAM instructions that correspond to a single Core-HUME expression. The resulting cost model is identical to that used in the resource inference that has been previously described in Deliverables D5 [JLH07b], D11 [JLH07a] and D14 [JLH07c].

$$\frac{}{E, h \vdash v \Downarrow_1 (v, h, \mathcal{R}^{\texttt{konst}})} \quad (\textsc{semValue})$$

$$\frac{E\ x = v}{E, h \vdash x \Downarrow_1 (v, h, \mathcal{R}^{\texttt{var}})} \quad (\textsc{semVar})$$

$$\frac{l = \text{freshloc (dom } h) \qquad \vec{rs} = E \star \vec{xs} \qquad h' = h(l \mapsto (c,\ \vec{rs}))}{E, h \vdash c\ \vec{xs} \Downarrow_1 (l, h', \mathcal{R}^{\texttt{constr}\ \#\vec{xs}})} \quad (\textsc{semConstr})$$

$$\frac{v_1 = E\ x_1 \qquad v_2 = E\ x_2 \qquad v = v_1 \oplus v_2}{E, h \vdash x_1 \oplus x_2 \Downarrow_1 (v, h, \mathcal{R}^{\oplus})} \quad (\textsc{semPrimBin})$$

$$\frac{E\ x = \text{true} \qquad E, h \vdash e_1 \Downarrow_n (v, h', p')}{E, h \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow_{n+1} (v, h', p' \frown \mathcal{R}^{\texttt{if true}})} \quad (\textsc{semIfTrue})$$

$$\frac{E\ x = \text{false} \qquad E, h \vdash e_2 \Downarrow_n (v, h', p')}{E, h \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow_{n+1} (v, h', p' \frown \mathcal{R}^{\texttt{if false}})} \quad (\textsc{semIfFalse})$$

$$\frac{E, h \vdash e_1 \Downarrow_m (v, h', p') \qquad E(x := v), h' \vdash e_2 \Downarrow_n (v'', h'', p'')}{E, h \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 \Downarrow_{m+n} (v'', h'', p' \frown p'' \frown \mathcal{R}^{\texttt{let}})} \quad (\textsc{semLet})$$

$$\frac{l = E\ x \qquad \texttt{MATCH}\ E, h \vdash p\ \texttt{at}\ l \Downarrow (E', v') \qquad v' \neq \bot \qquad E', h \vdash e_1 \Downarrow_n (v, h', p')}{E, h \vdash \texttt{case}\ x\ \texttt{of}\ p \to e_1\ \texttt{otherwise}\ e_2 \Downarrow_{n+1} (v, h', p' \frown \mathcal{R}^{\texttt{case true}})} \quad (\textsc{semCaseTrue})$$

$$\frac{l = E\ x \qquad \texttt{MATCH}\ E, h \vdash p\ \texttt{at}\ l \Downarrow (E', v') \qquad v' = \bot \qquad E, h \vdash e_2 \Downarrow_n (v, h', p')}{E, h \vdash \texttt{case}\ x\ \texttt{of}\ p \to e_1\ \texttt{otherwise}\ e_2 \Downarrow_{n+1} (v, h', p' \frown \mathcal{R}^{\texttt{case false}})} \quad (\textsc{semCaseFalse})$$

$$\frac{E' = E(\text{fst (funTab } f) := E \star \vec{xs}) \qquad E', h \vdash \text{snd (funTab } f) \Downarrow_n (v, h', p)}{E, h \vdash f\ \vec{xs} \Downarrow_{n+1} (v, h', p \frown \mathcal{R}^{f\ \#\vec{xs}})} \quad (\textsc{semCalfun})$$

$$\frac{\begin{array}{c} l = E\ x \qquad \text{Some}\ (f,\ i,\ \vec{rs}) = h\ l \qquad (\vec{as}, e) = (\text{funTab}\ f) \\ i + \#\vec{xs} = \#\vec{as} \qquad E' = E(\vec{as} := \vec{rs} +\!\!+ (E \star \vec{xs})) \qquad E', h \vdash e \Downarrow_n (v, h', p) \end{array}}{E, h \vdash x\ \vec{xs} \Downarrow_{n+1} (v, h', p \smile \mathcal{R}^{\text{ap true } \#\vec{xs}})} \text{ (semCallVarExact)}$$

$$\frac{\begin{array}{c} l = E\ x \qquad \text{Some}\ (f,\ i,\ \vec{rs}) = h\ l \qquad (\vec{as}, e) = (\text{funTab}\ f) \qquad i + \#\vec{xs} < \#\vec{as} \\ l' = \text{freshloc}\ (\text{dom}\ h) \qquad h' = h(l' \mapsto (f,\ (i + \#\vec{xs}),\ (\vec{rs} +\!\!+ (E \star \vec{xs})))) \end{array}}{E, h \vdash x\ \vec{xs} \Downarrow_1 (l', h', \mathcal{R}^{\text{ap false } \#\vec{xs}})}$$

(semCallVarUnderApp)

Discussion: A value v is directly returned as result. For a variable x a lookup is performed in the current environment E and the result is returned. In a constructor application $c\ \vec{xs}$ the values of the variables $\vec{xs}$ are used as arguments to the c and the heap is extended with a fresh binding to the constructor. For primitive operations a shallow embedding is used so that the operator $\oplus$ can be directly applied to the values of its variables. A conditional returns the value of the then branch e_1, if the value in the variable x is true, or the value of the else branch e_2, if the value in the variable x is false. A let binds the result of e_1 to the variable x and returns the value of the body e_2. A case expression is a one-step matching expression, that matches a variable x against a pattern p; if successful its value is that of e_1, if unsuccessful its value is that of e_2. We choose such a representation of cases, which requires nesting in the `otherwise` branch if a variable should be matched against several patterns, to simplify the formalisation. In first-order function calls, we bind the values in the arguments to the formal parameters ($\vec{as}$), and use the modified environment to evaluate the body (e) of the function f. The table funTab is a mapping of function names to pairs, with the argument list as first, and the function body as the second argument. For higher-order function calls, we have to distinguish two cases. In the situation that we have precisely the number of arguments that the function requires, the values of each of the arguments ($\vec{xs}$) are bound to the formal parameters ($\vec{as}$) of the function, before the function body is evaluated using these new bindings. In the situation that we have fewer than the required number of arguments, however, the result will be a new closure consisting of f paired with both its existing arguments ($\vec{rs}$) and all the new arguments. These are obtained by looking up the values of the variables, $\vec{xs}$, in the environment, E.

For readability, we also define the following semantic relation, which abstracts over the index n in the above relation (this index is needed for technical reasons in the proof).

Definition 1 (semantics) $E,\ h\ \vdash\ e\ \Downarrow\ (v, h', p)\ \equiv\ \exists n.\ E,\ h\ \vdash\ e\ \Downarrow n\ (v, h', p)$

3.3 Program Logic and Assertion Language

In this section we develop a VDM-style [Jon90] program logic for Core-HUME, and in doing so we define the format of assertions as predicates over the components of the program state. A judgement has the form $G\ \rhd\ e\ :\ P$, meaning that expression e fulfills the assertion P in a context G.

An *assertion* is a predicate over the components of the operational semantics, namely environment E, pre-heap h, post-heap hh, result value v and resources p. A context is a set of pairs of program expression $e \in \text{Expr}$ and assertion $P \in \text{Assn}$.

$$\begin{array}{rcl} P, Q\ \in\ \text{Assn} & = & \text{Env} \Rightarrow \text{Heap} \Rightarrow \text{Heap} \Rightarrow \text{Val} \Rightarrow \text{Resources} \Rightarrow \mathbb{B} \\ G\ \in\ \text{Ctxt} & = & \mathcal{P}\ \{(\text{Expr}, \text{Assn})\} \end{array}$$

The following set of rules defines the program logic for Core-HUME.

$$\frac{}{G\ \rhd\ w\ :\ \lambda E\ h\ hh\ v\ p.\ v = w\ \wedge\ h = hh\ \wedge\ p = \mathcal{R}^{\text{konst}}} \text{ (vdmValue)}$$

$$\frac{}{G \ \rhd \ x \ : \ \lambda E\ h\ hh\ v\ p.\ v = E\ x \ \wedge \ h = hh \ \wedge \ p = \mathcal{R}^{\texttt{var}}} \quad \text{(vdmVar)}$$

$$\frac{}{G \ \rhd \ c\ \vec{xs} \ : \ \lambda E\ h\ hh\ v\ p.\ \exists l\ \vec{rs}.\ l = \text{freshloc}(\text{dom}\ h) \ \wedge \ \vec{rs} = E \star \vec{xs} \ \wedge \ v = l \ \wedge \ hh = h(l \mapsto (c,\ \vec{rs})) \ \wedge p = \mathcal{R}^{\texttt{constr}\ \#\vec{xs}}} \quad \text{(vdmConstr)}$$

$$\frac{}{G \ \rhd \ x_1 \oplus x_2 \ : \lambda E\ h\ hh\ v\ p.\ \exists v_1\ v_2.\ E\ x_1 = v_1 \ \wedge \ E\ x_2 = v_2 \ \wedge \ v = v_1 \oplus v_2 \ \wedge \ h = hh \ \wedge \ p = \mathcal{R}^{\oplus}} \quad \text{(vdmPrimBin)}$$

$$\frac{G \ \rhd \ e_1 \ : P_1 \qquad G \ \rhd \ e_2 \ : \ P_2}{\begin{array}{c} G \ \rhd \ \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \ : \ \lambda E\ h\ hh\ v\ p. \\ ((E\ x = \text{true} \ \longrightarrow \ \exists p'.\ P_1\ E\ h\ hh\ v\ p' \ \wedge \ p = p' \smile \mathcal{R}^{\texttt{if true}}) \wedge \\ (E\ x = \text{false} \ \longrightarrow \ \exists p'.\ P_2\ E\ h\ hh\ v\ p' \ \wedge \ p = p' \smile \mathcal{R}^{\texttt{if false}})) \end{array}} \quad \text{(vdmIf)}$$

$$\frac{G \ \rhd \ e_1 \ : P_1 \qquad G \ \rhd \ e_2 \ : \ P_2}{G \ \rhd \texttt{let}\ x \ = \ e_1\ \texttt{in}\ e_2 \ : \ \lambda E\ h\ hh\ v\ p.\ \exists v'\ h'\ p'\ p''.\ (P_1\ E\ h\ h'\ v'\ p' \ \wedge \ P_2\ E(x := v')\ h'\ hh\ v\ p'' \ \wedge \ p = p' \smile p'' \smile \mathcal{R}^{\texttt{let}})} \quad \text{(vdmLet)}$$

$$\frac{\begin{array}{c} G \ \rhd \ e_1 \ : \ P_1 \qquad G \ \rhd \ e_2 \ : \ P_2 \end{array}}{\begin{array}{c} G \ \rhd \ \texttt{case}\ x\ \texttt{of}\ p_1 \rightarrow e_1\ \texttt{otherwise}\ e_2 \ : \ \lambda E\ h\ hh\ v\ p.\ \forall l.\ l = E\ x \ \longrightarrow \ \forall E'\ v'. \\ \texttt{MATCH}\ E,\ h \vdash \ p_1\ \texttt{at}\ l \ \Downarrow \ (E', v') \ \longrightarrow \\ ((v' = \bot \ \longrightarrow \ \exists p'.\ P_2\ E\ h\ hh\ v\ p' \ \wedge \ p = p' \smile \mathcal{R}^{\texttt{case false}}) \wedge \\ (v' \neq \bot \ \longrightarrow \ \exists p'.\ P_1\ E'\ h\ hh\ v\ p' \ \wedge \ p = p' \smile \mathcal{R}^{\texttt{case true}})) \end{array}} \quad \text{(vdmCase)}$$

$$\frac{\{(f\ \vec{xs}, P)\} \cup G \ \rhd \ \text{snd}\ (\text{funTab}\ f) \ : \ \lambda E\ h\ hh\ v\ p.\ \forall E'.\ E = E'(\text{fst}\ (\text{funTab}\ f) := E' \star \vec{xs}) \longrightarrow P\ E'\ h\ hh\ v\ (p \smile \mathcal{R}^{f\ \#\vec{xs}})}{G \ \rhd \ f\ \vec{xs} \ : \ P} \quad \text{(vdmCalfun)}$$

$$\frac{\begin{array}{c} \forall E\ h\ hh\ v.\ \Phi_{x,\vec{xs}}\ E\ h\ hh\ v \longrightarrow \ P\ E\ h\ hh\ v\ \mathcal{R}^{\texttt{ap false}\ \#\vec{xs}} \\ (\forall f.\ \{(x\ \vec{xs}, P)\} \cup G \ \rhd \ \text{snd}\ (\text{funTab}\ f) \ : \ \lambda E\ h\ hh\ v\ p. \\ \forall E'.\ (\exists\ \vec{rs}.\ \Psi_{x,\vec{xs}}\ E'\ h\ hh\ v\ f\ \vec{rs} \ \wedge E = E'(\text{fst}\ (\text{funTab}\ f) := \vec{rs} +\!\!+ (E' \star \vec{xs})) \ \longrightarrow \ P\ E'\ h\ hh\ v\ (p \smile \mathcal{R}^{\texttt{ap true}\ \#\vec{xs}})) \end{array}}{G \ \rhd \ x\ \vec{xs} \ : \ P} \quad \text{(vdmCallVar)}$$

$$\frac{(e, P) \in G}{G \ \rhd \ e \ : \ P} \quad \text{(vdmAx)}$$

$$\frac{\forall E\ h\ hh\ v\ p.\ P\ E\ h\ hh\ v\ p \ \longrightarrow \ Q\ E\ h\ hh\ v\ p \qquad G \ \rhd \ e \ : \ P}{G \ \rhd \ e \ : \ Q} \quad \text{(vdmConseq)}$$

$$\begin{array}{rcl} \Phi_{x,\vec{xs}} & \equiv & \lambda E\ h\ hh\ v.\ \exists l\ l'\ f\ i\ \vec{rs}.\ E\ x = l \ \wedge \ h\ l = \text{Some}(f,\ i,\ \vec{rs}) \ \wedge \ i + \#\vec{xs} < \#(\text{fst}\ (\text{funTab}\ f)) \ \wedge \\ & & l' = \text{freshloc}\ (\text{dom}\ h) \ \wedge \ v = l' \ \wedge \ hh = h(l' \mapsto (f,\ (i + \#\vec{xs}),\ (\vec{rs} +\!\!+ (E \star \vec{xs})))) \\ \Psi_{x,\vec{xs}} & \equiv & \lambda E'\ h\ hh\ v\ f\ \vec{rs}.\ \exists l\ i\ .\ E'\ x = l \ \wedge \ h\ l = \text{Some}\ (f,\ i,\ \vec{rs}) \ \wedge \ i + \#\vec{xs} = \#(\text{fst}\ (\text{funTab}\ f)) \end{array}$$

Discussion: The rules of the program logic directly encode the modifications to environment and heap, the result value and resource consumption as expressed in the operational semantics. A value w is the result value, and the heap is unchanged (VDMVALUE). A variable x requires a lookup in the environment, and again the heap is unchanged (VDMVAR). In a constructor application, the newly allocated location is existentially quantified, and the heap is updated with a binding to this location (VDMCONSTR). The two possible control flows in the conditional are encoded in the logic as implications, based on the boolean contents of variable x (VDMIF). The intermediate heap and resources need to be existentially quantified. Similarly, in the let rule (VDMLET), the intermediate value, heap and resources, that are produced by executing expression e_1 are existentially quantified, and the environment for executing e_2 is updated accordingly. The control flows in the case construct are encoded as implications, based on the result value from the pattern match (VDMCASE). In a first order call (VDMCALFUN), the assertion to be proven for f is added with the call into the context, and the function body has to fulfill the assertion, after modifying environment and resources. The higher-order function call (VDMCALLVAR) directly encodes the preconditions of the two cases of under application (Φ) and of exact application (Ψ) from the operational semantics. Thus, the definition of Φ reads as follows: for the pre-state E, h we find in variable x a closure with i arguments, and the total number of arguments is smaller than the function's arity ($i + \#\vec{xs} < \#(\text{fst (funTab } f))$). In this case, the result state v, hh, p is constructed such that the result value is a new closure, containing all arguments, and the result heap is updated accordingly. The definition of Ψ reads as follows: For the pre-state E, h we find in variable x a closure with i arguments, and the total number of arguments matches the function's arity ($i + \#\vec{xs} = \#(\text{fst (funTab } f))$). In this case, the second premise in rule VDMCALLVAR demands that the body of f (snd (funTab f)) fulfils the assertion P, with the environment adjusted for parameter passing and a modified resource vector to account for the costs of the function call. Note that Ψ has to be parameterised over the function f and the argument values $\vec{rs}$ captured in its closure, so that the VDMCALLVAR rule can quantifiy over f and $\vec{rs}$. In particular, the quantification over f scopes over the entire second judgement in the premise of VDMCALLVAR, because it is needed to retrieve the function's body via snd (funTab f). Finally, the quantification over $\vec{rs}$ scopes over the entire pre-condition inside the assertion used in the premise's judgement. The second component in this precondition states that the environment E' is constructed out of E by binding the values of the formal parameters of the function f to those found in the closure, $\vec{rs}$, and those retrieved from the arguments $\vec{xs}$ provided in the functional call. We could further refine the quantification over f, to capture only those functions, which have exactly $i + \#\vec{xs}$ arguments of matching type (if we further model types in the program logic and store them in the funTab as we do for formal parameters and function body). Such a version would make an application of the VDMCALLVAR rule simpler, since the range of the quantification is reduced. However, for readability of the already complex rule we do not include this optimisation here, but we plan to investigate it in the future. The final two rules, VDMAX and VDMCONSEQ, are the standard rules for using an axiom, present in the context, and for logical consequence in the meta-language.

3.4 Soundness

As the main result on the program logic in the previous section, we prove its soundness w.r.t. to the operational semantics.

First we define (semantic) validity of an assertion for an expression in a context. As for the definition of the operational semantics, it is parameterised with a counter n, as basis for the induction in the soundness proof. Since we use a shallow embedding of assertions, the validity statement is remarkably simple: the predicate, representing the assertion, has to be true for all pre- and post-states, which are related by the operational semantics relation.

Definition 2 (validity) *An assertion P is valid for expression e, written $\models_n e : P$ iff*

$$\forall m \leq n.\ \forall E\ h\ hh\ v\ p.\ (E,\ h\ \vdash\ e\ \Downarrow_m\ (v, hh, p))\ \longrightarrow\ P\ E\ h\ hh\ v\ p$$

Definition 3 (validity-in-context) *An assertion P is valid for expression e in context G, written $G \models e : P$ iff*

$$\forall n.\ (\forall (e', P')\ \in\ G.\ \models_n\ e'\ :\ P')\ \longrightarrow \models_n\ e\ :\ P$$

Based on these definitions, the soundness theorem can be stated as follows. The full details of the proof are given in the Isabelle/HOL formalisation.

Theorem 1 (soundness) *For all contexts G, expressions e, assertions P the following holds*

$$G \rhd e : P \Longrightarrow G \models e : P$$

Proof structure: By induction over the rules of the program logic. In the case of function calls, induction over the index n in the semantics relation. □

3.5 Examples of Resource Consumption

As an example of using our assertion language to express resource bounds, we examine the following Hume program, which computes the `sum` over a list of integer values, using a tail-recursive, accumulating function `sumAcc`:

```
sumAcc :: int 32 -> [int 32] -> int 32;
sumAcc acc []     = acc;
sumAcc acc (x:xs) = sumAcc (acc+x) xs;

sum :: [int 32] -> int 32;
sum xs = sumAcc 0 xs;
```

The intermediate Core-HUME code, as used in the compiler is shown below. Compared to the language we have formalised in this section, it has several annotations that are used by later compiler phases, and it introduces several variants of basic constructs. For example, `fcase` is a variant of `case`, which indicates that it is the top-level match in a function. It is annotated with the maximum number of variables used in any of the patterns. The left-to-right order of pattern matches in Hume becomes explicit with the nested `case`s in Core-HUME, and the rule-by-rule order remains explicit in the top-level-case. Given this intermediate code representation, that is already used in the compiler, it is easy to construct a version of the program that can be used as input to the Isabelle/HOL formalisation as discussed in this document.

```
{{-## FUND (non-lifted) sumAcc :: int,(?list1int int) ->int  int  ?list1int-}
 sumAcc :: (int, ?list1int -> int) (?sumAcc_arg_11  :: int)  (?sumAcc_arg_12  :: ?list1int) =
 fcase 3 ?sumAcc_arg_11  of
   (acc ) -> fcase ?sumAcc_arg_12  of
               ((?N1int) ) -> glet ?bdg_acc_7 = acc
                              in ?bdg_acc_7
             esac |
   (acc ) -> fcase  ?sumAcc_arg_12  of
               (?C1int x   xs  ) -> glet ?bdg_xs_3 = xs ;
                                         ?bdg_x_5 = x ;
                                         ?bdg_acc_6 = acc ;
                                         ?z_4 = (?bdg_acc_6  +.i ?bdg_x_5 )
                                    in (sumAcc $! ?z_4   ?bdg_xs_3  )
             esac
 esac ;
 {-## FUND (non-lifted) sum :: (?list1int int) ->int  ?list1int-}
 sum :: (?list1int -> int) (?sum_arg_11  :: ?list1int) =
 fcase  1 ?sum_arg_11  of
```

```
   (xs ) -> glet ?bdg_xs_1 = xs ;
                 ?z_2 = 0
            in (sumAcc $$ ?z_2   ?bdg_xs_1 )
 esac }
```

Heap consumption

Performing an analysis of the heap space usage yields the following enriched types for the functions defined in this program:

```
sumAcc :: 0, (int<0>,?list1int[?C1int:2;int<0>,#|?N1int:0]) -(0/0)-> int<0> ,0
sum    :: 0, (?list1int[?C1int:2;int<0>,#|?N1int:0]) -(2/0)-> int<0> ,0
```

In particular, these types specify a linear upper bound on the heap consumption of the `sum` function: let n be the length of the list passed to `sum`; then the heap consumption is bounded by $2n + 2$.
We can now translate this bound into a resource statement in our assertion language:

$$G = \{((sumAcc\ acc\ xs)\ ,\ \lambda E\ h\ h'\ v\ p.\ E\ xs = l\ \wedge\ h,\ l \models_{List} n\ \longrightarrow |\ \mathrm{dom}\ h'\ | \leq |\ \mathrm{dom}\ h\ | + (2n + 0))\}$$
$$\forall l\ n.\ G \rhd (sum\ xs)\ :\ \lambda E\ h\ h'\ v\ p.\ E\ xs = l\ \wedge\ h,\ l \models_{List} n\ \longrightarrow |\ \mathrm{dom}\ h'\ | \leq |\ \mathrm{dom}\ h\ | + (2n + 2)$$

G contains the "resource specifications" of all functions in the program, using the same assertion format as for the `sum` function above. The function dom returns the domain of a partial map, and the function $|\cdot|$ returns the size of the set. Heap consumption is thus formalised as the difference between the sizes of the domains in the post- and pre-heap. We use a predicate $h, l \models_{List} n$ to specify that in heap h at location l we find a list-structure of length n. For every algebraic data type we inductively define such a predicate. In a fully integrated system, the compiler can automatically generate such a predicate. Hume-level lists are treated as algebraic data types in Core-HUME, with constructors `?C1int` and `?N1int`.

Stack consumption

Performing an analysis of the stack space usage yields the following enriched types for the functions defined in this program:

```
sumAcc :: 1, (int<0>,?list1int[?C1int:0;int<0>,#|?N1int:0]) -(10/9)-> int<0> ,0
sum    :: 1, (?list1int[?C1int:0;int<0>,#|?N1int:0]) -(17/16)-> int<0> ,0
```

In particular, these types specify that the total stack space consumption of `sumAcc` is bounded by 10 and of `sum` is bounded by 17. Note that we obtain constant bounds, since the compiler detects the tail call in `sumAcc` and therefore can reuse the function frame.

We can now translate this bound into a resource statement in our assertion language:

$$G = \{((sumAcc\ acc\ xs)\ ,\ \lambda E\ h\ h'\ v\ p.\ E\ xs = l\ \wedge\ h,\ l \models_{List} n\ \longrightarrow\ maxstack\ p \leq 10)\}$$
$$\forall l\ n.\ G \rhd (sum\ xs)\ :\ \lambda E\ h\ h'\ v\ p.\ E\ xs = l\ \wedge\ h,\ l \models_{List} n\ \longrightarrow\ maxstack\ p \leq 17$$

Again, G contains the "resource specifications" of all functions in the program, using the same assertion format as for the `sum` function above.

Worst-case execution time

Performing an analysis of the worst-case execution time yields the following enriched types for the functions defined in this program:

```
sumAcc :: 39, (int<0>,?list1int[?C1int:746;int<0>,#|?N1int:0]) -(427/262)-> int<0> ,0
sum    :: 39, (?list1int[?C1int:746;int<0>,#|?N1int:0]) -(685/93)-> int<0> ,0
```

In particular, these types specify a linear upper bound on the worst-case execution time for both functions. Let n be the length of the input list for `sumAcc` and `sum`, respectively. Then, the execution time for `sumAcc` is bounded by $746n + 427$ and the execution time for `sum` is bounded by $746n + 685$. Unsurprisingly, the difference in the bounds is only a constant factor, covering the setup costs in `sum` to start the recursion.

We can now translate this bound into a resource statement in our assertion language:

$$G = \{((sumAcc\ acc\ xs)\ ,\ \lambda E\ h\ h'\ v\ p.\ E\ xs = l\ \wedge\ h,\ l \models_{List} n\ \longrightarrow\ clock\ p \leq 746n + 427)\}$$
$$\forall l\ n.\ G \rhd (sum\ xs)\ :\ \lambda E\ h\ h'\ v\ p.\ E\ xs = l\ \wedge\ h,\ l \models_{List} n\ \longrightarrow\ clock\ p \leq 746n + 685$$

Again, G contains the "resource specifications" of all functions in the program, using the same assertion format as for the `sum` function above.

4 An Assertion Language for Coordination-level Hume

In this section we introduce an assertion language for the coordination-level of Hume.

4.1 Source Language

Coordination-level Hume describes a network of boxes, synchronising and communicating over wires. A wire is a single value buffer that connects two boxes. Thus a box will block with pending output until all of its required output wires become empty. The Hume execution model is based on cyclical execution, where each cycle has two phases: in the first phase each box is run once and attempts to consume inputs and generate outputs; and in the second phase the output changes are resolved in a unitary super-step. At the end of each phase a box may be in one of the following states:

Definition 4 *Scheduling states*

1. **Runnable** *The box has successfully consumed inputs and asserted outputs.*
2. **Blocked** *The box has successfully consumed inputs but failed to assert outputs. It will attempt to assert outputs on subsequent cycles.*
3. **Matchfail** *The box has failed to match required inputs.*

Hume uses a *lock-step* scheduling mechanism, which works as follows:

Definition 5 *Lock-step scheduling*

for ever
  for each ***Runnable*** *and* ***Matchfail*** *box*
    execute (box)
  super_step (each ***Runnable*** *and* ***Blocked*** *box)*

In the "execute" phase boxes are executed separately while the "super-step" phase is one distinct step. Executing a box works as follows:

execute (box) $\triangleq$ *let match* $\triangleq$ *pattern_match*$\big($*inputs(box),matches(box)*$\big)$ *in*
if match $= \perp$ *inputs(B) return input wires of box B*
then set ***Matchfail*** *state matches(B) returns B's list of matches*
else evaluate$\big($*expression(match)*$\big)$ *and set* ***Runnnable*** *state*

where *expression(M)* returns the expression of match M and *evaluate(E)* computes the output of expression E. *pattern_match* works as follows:

pattern_match (inputs,match) $\triangleq$ *patterns(X) returns pattern of match X*
if pattern_matches $\big($*patterns(match),inputs*$\big)$ *A single pattern is matched*
then consume inputs and return match return succeeded match
else if exists$\big($*next(match)*$\big)$ *exists(X) holds if a match succeeds X*
then pattern_match$\big($*inputs, next(match)*$\big)$ *next(X) is the next match*
else return $\perp$ *No more matches*

Finally, the *super_step* for a box has the following functionality:

super_step (box) $\triangleq$
if all output wires are asserted Required output wires are empty
then update wire value and set ***Runnable*** *state*
else set ***Blocked*** *state Output values dangling*

4.2 Semantics

The scheduling has been formalised in Lamport's *Temporal Logic of Actions* (TLA) [Lam94]. It allows us to reason about the coordination layer in a more abstract way than using the structural operational semantics directly.

TLA has a similar design to Hume. It builds a temporal logic on top of an action layer, in the same way as the coordination layer is on top of the functional layer in Hume. It is untyped logic, and assumes an infinite set Var of variables and an infinite set Val of values. Since TLA is untyped, tuples and lists are represented in the same way, and we use $\langle \cdots \rangle$ as a notation. Further l_i is the i^{th} element of list l, i.e. the accessor function of list l. Although, it is a list and not a function we sometimes used λ notation for lists. $a \cdot b$ appends list b to the end of list a. We will use the following list functions:

$$\begin{aligned} hd(\langle x, y, z, \cdots \rangle) &\triangleq x \\ tl(\langle x, y, z, \cdots \rangle) &\triangleq \langle y, z, \cdots \rangle \\ len(ls) &\triangleq \textbf{if } ls = \langle\rangle \textbf{ then } 0 \textbf{ else } 1 + len\big(tl(ls)\big) \end{aligned}$$

In the following discussion we use natural numbers to both identify boxes and wires. We therefore have two sets:

$$\begin{aligned} Boxes &\subseteq \mathbb{N} \\ Wires &\subseteq \mathbb{N} \end{aligned}$$

and unless otherwise specified n is the largest element of *Boxes* and m of *Wires*. Boxes are actions while wires are one-buffered communications links, i.e. each variable is shared by two boxes. We will use w for the list of all wires, and a distinct wire $j \in Wires$ is accessed by w_j. Note the element w_j of the list of wires is a distinct variable. Each box $i \in Boxes$ will have a tuple of input wires iws_i and output wires ows_i. The wires are only accessed through this, i.e. they are only accessed from within the boxes, except for the case that the wire is connected to an external device, which is represented by a dedicated action. However, such *open systems* are ignored here.

env is a list of the internal environments of all boxes, meaning n elements, where the element env_i of box $i \in Boxes$ consists of the following elements:

$$env_i \triangleq \langle state_i, res_i, \mathcal{V}_i, m_i \rangle$$

This means that *state*,*res*, $\mathcal{V}$ and m are lists (of length n) which can be accessed directly. Again, all the elements of all these lists are distinct variables in the TLA formalisation. $state_i$ is the scheduling state of box i as explained in the previous section. res_i is the result of executing the expression of a box (its output buffer). $\mathcal{V}_i \subseteq (\mathrm{Var} \mapsto \mathrm{Val})$ is the environment of a box, holding the bound variables.

For the semantics of the expression-layer of Hume we refer to Section 3. Here, we simply use the function *execute* for evaluating an expression e in an environment $\mathcal{V}$. Using the operational semantics notation of the expression level from the previous section, we have $\mathcal{V}, \emptyset \vdash e \Downarrow (execute(\mathcal{V}, e), h, p)$. Furthermore, we can use properties that have already been verified on expression level as axioms for the verification on the coordination level.

In some cases, it is useful to construct a formal relation with all the variables associated with a box, i.e. the environment, *env*, in addition to the input and output wires. This environment is represented by *comenv*. In addition, we define $unch(i)$ to leave all the variables that are owned by box i unchanged. This depends on the current scheduling step.

$$
\begin{array}{rcl}
comenv_i & \triangleq & \langle iws_i, ows_i \rangle \cdot env_i \\
unch(i) & \triangleq & env'_i = env_i \\
& \wedge & (sch = Consume \Rightarrow iws'_i = inw_i) \wedge\ (sch = Super \Rightarrow ows'_i = ows_i)
\end{array}
$$

Each box i has a ruleset rs_i which is a list of pattern and expression pairs. To keep it close to Hume syntax we use the normal $p \rightarrow e$ notation. We define projection functions for the two elements:

$$
\begin{array}{rcl}
rs_i & \triangleq & \langle (p_1 \rightarrow e_1), \cdots, (p_n \rightarrow e_n) \rangle \\
patt(p \rightarrow e) & \triangleq & p \\
exp(p \rightarrow e) & \triangleq & e
\end{array}
$$

$$
\begin{array}{rcl}
pmatch(p, iw) & \triangleq & p \in \{\texttt{\_*}, \texttt{*}\} \vee (p \in \{\texttt{x}, \texttt{\_}\} \wedge iw \neq \bot) \vee (p = iw) \quad \boxed{\texttt{x} \in \mathrm{Var}} \\
pattmatch(patt, iws) & \triangleq & \textbf{if } iws = \langle\rangle \\
& & \quad \textbf{then } True \\
& & \quad \textbf{else } pmatch\big(hd(patt), hd(iws)\big) \wedge pattmatch\big(tl(patt), tl(iws)\big) \\
matching(n, rs, iws) & \triangleq & \textbf{if } rs = \langle\rangle \\
& & \quad \textbf{then undef} \\
& & \quad \textbf{else if } pattmatch\Big(patt\big(hd(rs)\big), hd(iws)\Big) \\
& & \qquad \textbf{then } n + 1 \\
& & \qquad \textbf{else } matching\big(n + 1, tl(rs), tl(iws)\big) \\
match(rs, iws) & \triangleq & matching(0, rs, iws)
\end{array}
$$

$$
\begin{array}{rcl}
consume(rs, iws) & \triangleq & \textbf{if } iws = \langle\rangle \\
& & \quad \textbf{then } \langle\rangle \\
& & \quad \textbf{else if } hd(rs) \neq \bot \\
& & \qquad \textbf{then } \langle\bot\rangle \cdot consume\big(tl(rs), tl(iws)\big) \\
& & \qquad \textbf{else } \langle hd(iws)\rangle \cdot consume\big(tl(rs), tl(iws)\big) \\
bind(\mathcal{V}, rs, iws) & \triangleq & \textbf{if } iws = \langle\rangle \\
& & \quad \textbf{then } \mathcal{V} \\
& & \quad \textbf{else if } hd(rs) = \texttt{x} \quad \boxed{\texttt{x} \in \mathrm{Var}} \\
& & \qquad \textbf{then } bind\Big(\big(\lambda y.\ \textbf{if } \texttt{x} = y \textbf{ then } hd(iws) \textbf{ else } \mathcal{V}\, y\big), tl(rs), tl(iws)\Big) \\
& & \qquad \textbf{else } bind\big(\mathcal{V}, tl(rs), tl(iws)\big)
\end{array}
$$

Figure 3: Pattern matching, consuming inputs and binding variables

Absence of a value for a variable in a box is represented by '`undef`'. For a wire this is represented as '$\bot$'. '`undef`' and '$\bot$' have the same semantics, but there are many situations where it is useful to distinguish them formally.

$$
\begin{array}{rcl}
\mathit{assertOut}(\mathit{res}, \mathit{ows}) & \triangleq & \mathit{ows} = \bot \vee \mathit{res} = \bot \\
\mathit{assertOuts}(\mathit{res}, \mathit{ows}) & \triangleq & \textbf{if } \mathit{ows} = \langle\rangle \\
 & & \quad \textbf{then } \mathit{True} \\
 & & \quad \textbf{else } \mathit{assertOut}\big(\mathit{hd}(\mathit{res}), \mathit{hd}(\mathit{ows})\big) \wedge \mathit{assertOuts}\big(\mathit{tl}(\mathit{res}), \mathit{tl}(\mathit{ows})\big) \\
\mathit{write\_el}(\mathit{res}, \mathit{ow}) & \triangleq & \textbf{if } \mathit{res} \neq \bot \textbf{ then } \mathit{res} \textbf{ else } \mathit{ow} \\
\mathit{write}(\mathit{res}, \mathit{ows}) & \triangleq & \textbf{if } \mathit{ows} = \langle\rangle \\
 & & \quad \textbf{then } \langle\rangle \\
 & & \quad \textbf{else} \Big\langle \mathit{write\_el}\big(\mathit{hd}(\mathit{res}), \mathit{hd}(\mathit{ows})\big) \Big\rangle \cdot \mathit{write}\big(\mathit{tl}(\mathit{res}), \mathit{tl}(\mathit{ows})\big)
\end{array}
$$

Figure 4: Asserting and Writing Outputs

In a pattern/expression '$*$' denotes ignore, '_' in a pattern means presence of a value, and '_$*$' combines them, i.e. if a value is present behave like '_' else behave like '$*$'. This is the same syntax as in the Hume source code.

Pattern matching is formalised in TLA by *match(rs,iws)* in Figure 3. It returns the succeeding match number, and `undef` if all matches fails. This number is to execute the correct expression. $\mathit{consume}(p, \mathit{iws})$ returns the new values on the input variables after a match, while $\mathit{bind}(p, \mathit{iws})$ binds the required variables to the environment.

Figure 4 shows the actions for asserting and writing the outputs. Note that for a given value it succeeds if either the wire is empty or the corresponding element of the output buffer is empty.

Figure 5 shows the TLA actions for a Hume program. The action here uses the previously defined actions. For simplicity, we have refined the previously two-phased lock-step scheduling model into three phases, by splitting the *Execute* phase into a *Consume* and *Compute* phase. The *Consume* step matches and consumes the inputs while the *Compute* step computes the result. This is simply adding detail to the higher-level definition there. Furthermore, according to the structured operational semantics boxes are, in this phase, executed one-by-one. In this phase the variables updated are disjoint for all boxes. Hence, it doesn't matter if we look at this as one step or one step per box. While the former reduces the size of the proofs in general, it complicates the *Compute* phase in the sense that extra work is needed for transformations. We have therefore chosen to represent the *Consume* phase as one distinct phase, and the *Compute* phase as one for each box.

The variable *sch* keeps track of the current scheduling step and is updated by the scheduler $\mathcal{S}$. B is a set holding all the boxes that have not been scheduled in the *Compute* step. The scheduler cannot proceed to the *Super* step until this set is empty, i.e. all boxes have executed.

The action for a box i is represented by $\mathcal{B}_i$ which consists of a sub-action for each of the scheduling steps. This embedding differs from the lock-step scheduling in the way that we always attempt to execute the boxes, regardless of their state. The box action then decides if any computation is required. The reason behind this is that in TLA we must explicitly state that a variable is left unchanged. Moving the test to the box provides a more readable and elegant solution. v holds all variables used to specify the whole Hume program. $\mathcal{N}$ is the complete next-action for a Hume program[1]. In the *Compute* step the wires are left unchanged, hence $w' = w$. All variables belonging to the boxes not being executed are also left unchanged. If we wanted to specify an open system, we also had to define a simulator updating wires to external devices in the *Consume* and *Super* phase.

The initial state is given as follows:

$$
\begin{array}{rcl}
\mathit{Init} \triangleq & & B = \mathit{Boxes} \\
 & \wedge & w = k \\
 & \wedge & \mathit{sch} = \mathit{Consume} \\
 & \wedge & \bigwedge_{i \in \mathit{Boxes}} \langle \mathit{state}_i, \mathit{res}_i, \mathcal{V}_i, m_i \rangle = \langle \mathit{Runnable}, \texttt{undef}, \lambda x.\texttt{undef}, \texttt{undef} \rangle
\end{array}
$$

[1]Here we assume sequential execution of the boxes in the *Execute* step. This could also been executed in parallel.

$$
\begin{array}{rcl}
\mathcal{S} & \triangleq & (sch = Consume \Rightarrow sch' = Compute) \\
& \wedge & (sch = Compute \wedge B' = \{\} \Rightarrow sch' = Compute) \\
& \wedge & (sch = Compute \wedge B' \neq \{\} \Rightarrow sch' = Super) \\
& \wedge & (sch = Super \Rightarrow sch' = Consume)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{B}_i^1 & \triangleq & \textbf{if } state_i \neq Blocked \\
& & \textbf{then let } k = match(rs_i, iws_i) \textbf{ in} \\
& & \quad \textbf{if } k = \texttt{undef} \\
& & \quad \textbf{then} \langle \mathcal{V}_i, m_i, iws_i, state_i \rangle' = \langle \mathcal{V}_i, \texttt{undef}, iws_i, Matchfail \rangle \\
& & \quad \textbf{else } \langle m_i, state_i \rangle' = \langle k, Runnable \rangle \\
& & \quad \wedge\ iws' = consume\Big(patt\big((rs_i)_k\big), iws_i\Big) \\
& & \quad \wedge\ \mathcal{V}_i' = bind\Big(\mathcal{V}_i, patt\big((rs_i)_k\big), iws_i\Big) \\
& & \textbf{else } unch(i) \\
\mathcal{B}_i^2 & \triangleq & \textbf{if } state_i = Runnable \\
& & \textbf{then } res_i' = execute\Big(\mathcal{V}_i, exp\big((rs_i)_{m_i}\big)\Big) \\
& & \quad \wedge \langle m_i, \mathcal{V}_i, state_i \rangle' = \langle m_i, \mathcal{V}_i, state_i \rangle \\
& & \textbf{else } env_i' = env_i \\
\mathcal{B}_i^3 & \triangleq & \textbf{if } assertOuts(res_i, ows_i) \\
& & \textbf{then } res_i' = \texttt{undef} \ \wedge\ \mathcal{V}_i' = \lambda x.\texttt{undef} \ \wedge\ m_i' = \texttt{undef} \\
& & \quad \wedge\ ows_i' = write(res_i, ows_i) \\
& & \quad \wedge\ state_i' = Runnable \\
& & \textbf{else } state_i' = Blocked \\
& & \quad \wedge \langle res_i, \mathcal{V}_i, m_i, ows_i \rangle' = \langle res_i, \mathcal{V}_i, m_i, ows_i \rangle
\end{array}
$$

$$
\mathcal{B}_i \triangleq (sch = Consume \Rightarrow \mathcal{B}_i^1) \wedge (sch = Compute \Rightarrow \mathcal{B}_i^2) \wedge (sch = Super \Rightarrow \mathcal{B}_i^3)
$$

$$
\begin{array}{rcl}
v & \triangleq & \cdot w \cdot env \cdot \langle sch, B \rangle \\
\mathcal{N} & \triangleq & \left(\begin{array}{ll}
\wedge sch = Consume \vee sch = Super & \Rightarrow B' = Boxes \wedge \bigwedge_{i \in Boxes} \mathcal{B}_i \\
\wedge sch = Compute & \Rightarrow w' = w \wedge \exists i \in B : \left(\begin{array}{l} \wedge\ \mathcal{B}_i \wedge B' = B - \{i\} \\ \wedge\ \forall k \in Boxes : k \neq i \Rightarrow unch(k) \end{array} \right)
\end{array} \right) \wedge \mathcal{S}
\end{array}
$$

Figure 5: TLA Actions in Hume formalisation

Since a wire can be given an initial value in a Hume program we cannot simply assume that all of them are empty. Hence we assume a tuple of values k. The particular value of k is independent of the scheduling strategies and is therefore irrelevant for this particular discussion.

The only state in a Hume program should be the values on the wires. Consequently, the scheduler *sch*, the boxes to be scheduled B, and the internal state of the boxes *env* are variables required for the embedding. These are therefore hidden with the $\exists$ operator. As in a standard monolithic TLA specification, the full specification is a conjunction of the initial state and the next action prefixed by $\Box$ and written in a stuttering invariant way ($[\mathcal{N}]_v$):

$$
H \triangleq \exists\ env, sch, B : Init \wedge \Box[\mathcal{N}]_v \tag{1}
$$

4.2.1 Liveness

Until now, we have only discussed the safety aspects of a specification. A safety property states that something bad never happens, and (1) is sufficient to prove such properties. However, in such specifications we do not require that anything actually happens. *Liveness properties* assert that something good will eventually occur. The specification must therefore be constrained to remove such non-progress be-

haviours. This is achieved with a type of liveness call *fairness*. There are two types of fairness, and both of them depend on an *enabled* predicate: An action is *enabled* when it can be successfully executed. *Weak* fairness asserts that if an action remains enabled, then it will eventually execute, while *strong* fairness asserts that if an action is enabled infinitely often then it will eventually execute. Strong fairness is therefore strictly stronger then weak fairness. The scheduling of Hume guarantees both strong and weak fairness of boxes, since all boxes that can be executed will eventually be executed. Further, since the only way an executable (enabled) box can become in-executable (disabled) is to execute it, weak and strong fairness are equivalent for Hume. For this discussion, it is sufficient to consider a a weak fairness predicate on the next action $\mathcal{N}$. Thus, (1) is extended to

$$H^l \triangleq \exists\, env, sch, B : Init \wedge \Box[\mathcal{N}]_v \wedge WF_v(\mathcal{N}) \tag{2}$$

4.2.2 Real-time

The last type of properties is real-time properties, which requires the introduction of a timer. This will be discussed in Section 4.3.3.

4.3 Coordination-layer Properties and the Assertion Language

In this layer the properties refer to wires, i.e. we cannot access the internal parts of a box. To follow standard Hume notation, we refer to wires by their wire declaration. Hence, a wire w can can be referred to by either using the source or destination box/variable couple, identified by `<boxid>.<varid>`.

Before introducing the assertion language, we will first discuss the types of properties which the language will encapsulate, and, as in the previous section, we separate between safety, liveness and real-time properties.

4.3.1 Safety Properties

A safety property specifies that something bad never happens, i.e. it is an *invariant* of the system, which is specified with the always $\Box$ operator in temporal logic. With this we can for example specify that a property $P\ w$ always[2] holds for wire w.

Another safety property is that some action always hold. This will be explained by example below.

4.3.2 Liveness Properties

The first liveness property we consider, is a weaker version of the "wire invariant" above. Sometimes, the property $\Box(P\ w)$ is too strong. It might be sufficient to show that at some point in the future Pw holds. This is written $\Box\Diamond(P\ w)$ and is read as "always eventually".

The second liveness property is the *leads-to* property. It states that if $P\ w_1$ holds (for wire w_1) then eventually $Q\ w_2$ holds (for wire w_2). It is written $(P\ w_i) \rightsquigarrow (Q\ w_2)$, and abbreviates $\Box((P\ w_i) \Rightarrow \Diamond(Q\ w_2))$.

4.3.3 Real-time Properties

We are interested in one type of real-time properties called *bounded leads-to*. It states that if $P\ w_1$ holds (for wire w_1) then eventually $Q\ w_2$ holds (for wire w_2), *within* T time units. This property depends on properties of the expression layer, as stated in Section 3, although this discussion is independent of them. It will become clear where such analysis are required.

[2]We can distinguish between the assertion that a wire always has the given value, and properties that a wire always has the given value unless it is empty. Here, we refer to the latter of these.

For each such property we introduce a timer t to the specification. Initially, t equals *Disabled* (remember TLA is untyped). t is then updated by an action NT as follows:

- If t is *Disabled* and not $(P\ w_1)$ then t' remains *Disabled*
- If t is *Disabled* and $(P\ w_1)$ then t' is set to T
- If t is not *Disabled* and not $(Q\ w_2)$ then t' is decremented based on the property from the expression layer in the *Compute* step, and based on a constant of the *Consume* and *Super* step, which identifies the time it takes to update variables.
- If t is not *Disabled* and $(Q\ w_2)$ then t' is set to *Disabled*.

The specification of a program with one counter t is then:

$$H^{rt} \triangleq \exists\ env, sch, B, t : Init \wedge t = Disabled \wedge \Box[\mathcal{N} \wedge t' = NT(t)]_v \wedge WF_v(\mathcal{N}) \wedge WF_t(t' = NT(t)) \quad (3)$$

We must then prove the safety property $\Box(t \geq 0)$

4.3.4 The Assertion Language

The specification below encapsulates all the above properties. Furthermore, it contains a normal propositional calculus which combines different properties, as well as standard and defined Hume operators/functions that can be used.

In some cases it is hard to express the properties using the wires. For example, it might be easier to express properties via an external device which is updated by our Hume program. An example of this is given in the next section. The language therefore includes features for such auxiliary definitions. The BNF-grammar for the specification language `<Spec>` then becomes:

```
<prop> ::= <boxid>.<varid>      <Spec> ::= Inv <prop>
         |  Aux <id>                     |  Action <prop> <var>
         |  <prop>'                      |  <prop> Leadsto <prop>
         |  Not <prop>                   |  AlwEvent <prop>
         |  <prop> Or <prop> ...         |  IF <prop> THEN <prop> WITHIN <time>
         | <prop> + <prop> ...           |  AuxDef <id> = <boxid>.<varid> [Init <val>]
```

`Inv` and `Action` are the two safety properties, while `Leadsto` and `AlwEvent` are the obvious liveness properties. `IF...` defines the bounded leads-to property, while `AuxDef` is used to introduce an auxiliary definition as explained above. In `<prop>`, which is used by `<Spec>`, `<boxid>.<varid>` is used to refer to a wire; `Aux <id>` is used to refer to an auxiliary definition; `<prop>'` is used inside an action to refer to results (as is standard in TLA); it has all propositional operators like `Not` and `Or`; and finally, all standard and defined Hume operators like `+` and `==` are supported.

4.4 Examples

In this section we will illustrate the specification language with a simple example simulating a junction with two traffic-lights and a controller, represented by three boxes: two traffic light signals, `light1` and `light2` which are wired to their respective lights, which here are simply standard output. The `controller` box informs the lights when to change. The transitions between light patterns follows those for standard UK traffic lights. The Hume source code for the program is:

```
type Bit = word 1;
type lstate = (Bit,Bit);
type cstate = (Bit,Bit,Bit);
```

```
type tolight = cstate;

stream l1 to "std_out";
stream l2 to "std_out";

template light
in  (signal::Bit, state::lstate)
out (state'::lstate,lights::(tolight,string))
match
  (1,(0,0)) -> ((0,1),((1,1,0),"\n"))
| (1,(0,1)) -> ((1,0),((0,0,1),"\n"))
| (1,(1,0)) -> ((1,1),((0,1,0),"\n"))
| (1,(1,1)) -> ((0,0),((1,0,0),"\n"));

instantiate light as light * 2;

box controller
in (state::cstate)
out (state'::cstate, light1,light2::Bit)
match
  (0,0,0) -> ((0,0,1),1,*)  -- L1: Red -> Red-amber
| (0,0,1) -> ((0,1,0),1,*)  -- L1: Red-amber -> Green
| (0,1,0) -> ((0,1,1),1,*)  -- L1: Green -> Amber
| (0,1,1) -> ((1,0,0),1,*)  -- L1: Amber -> Red
| (1,0,0) -> ((1,0,1),*,1)  -- L2: Red -> Red-amber
| (1,0,1) -> ((1,1,0),*,1)  -- L2: Red-amber -> Gree
| (1,1,0) -> ((1,1,1),*,1)  -- L2: Green -> Amber
| (1,1,1) -> ((0,0,0),*,1); -- L2: Amber -> Red

wire controller (controller.state' initially (0,0,0))
                (controller.state,light1.signal,light2.signal);
wire light1 (controller.light1,light1.state')
            (light1.state initially (0,0),l1);
wire light2 (controller.light2,light2.state')
             (light2.state initially (0,0),l2);
```

4.5 Auxiliary Definitions

In this example the actual values/colours of the lights are interesting, and not the wires of the values. Thus, we introduce two auxiliary definitions `L1` and `L2`, which represent the actual trafficlights:

```
AuxDef L1 = Out light1.lights Init (1,0,0)
AuxDef L2 = Out light2.lights Init (1,0,0)
```

Note that, `L1` and `L2` will hold the last value on wires `light1.lights` and `light2.lights`, that is their values change when the values on these wires change[3].

[3]However, they do not change when the wires are set to `undef` – which makes it is easier to use for property specification

4.6 Safety properties

Firstly, we want to show that the lights change their values in the correct order. To verify this we must compare two states, i.e. a before light will have a given after light. We define the following function in the Hume source code:

```
Next l = case l of (1,0,0) -> (1,1,0)
                |  (1,1,0) -> (0,0,1)
                |  (0,0,1) -> (0,1,0)
                |  (0,1,0) -> (1,0,0);
```

This function returns the correct "new colour", given a "current colour" of the light. The order property is then specified as

```
Action ((Aux L1)' = Next(Aux L1)) L1
Action ((Aux L2)' = Next(Aux L2)) L2
```

which will be represented as follows in TLA:

$$\Box[L1' = \texttt{Next}\ L1]_{L1}$$
$$\Box[L2' = \texttt{Next}\ L2]_{L2}$$

The second safety property asserts that both lights cannot be green at the same time. This is an invariant, and is specified by:

```
Inv (Not((Aux L1) = (0,0,1)) Or (Not((Aux L2) = (0,0,1))))
```

Which is formalised in TLA as:

$$\Box(L1 \neq (0,0,1) \vee L2 \neq (0,0,1)) \tag{4}$$

This property is often not strong enough, since it is perfectly legal to drive when the light is amber. Hence, we strengthen this property to asserting that if one of the lights is not red, then the other light is red:

```
Inv ((Not((Aux L1) = (1,0,0)) Implies (Aux L2) = (1,0,0))
       And  (Not((Aux L2) = (1,0,0)) Implies (Aux L1) = (1,0,0)))
```

This is defined as follows in TLA:

$$\Box((L1 \neq (1,0,0) \Rightarrow L2 = (1,0,0)) \wedge (L2 \neq (1,0,0) \Rightarrow L1 = (1,0,0))) \tag{5}$$

4.6.1 Liveness properties

The first liveness property we show is that at any given time, there will be a time in the future when the lights are green. This is an "always eventually" property, defined as follows:

```
AlwEvent ((Aux L1) = (0,0,1)) And AlwEvent ((Aux L2) = (0,0,1))
```

and formalised as follows in TLA:

$$\Box\Diamond(L1 = (0,0,1)) \wedge \Box\Diamond(L2 = (0,0,1)) \tag{6}$$

Another liveness property is the leads-to property. For example, we can verify that if `L1` is red then `L2` will eventually become green:

```
((Aux L1) = (1,0,0)) Leadsto ((Aux L2) = (1,0,0))
```

which is formalised in TLA as:

$$L1 = (1,0,0) \rightsquigarrow L2 = (0,0,1) \tag{7}$$

4.6.2 Real-time properties

We have already shown that both lights will eventually be green. However, eventually may be a very long time. Ideally, we would like a maximum waiting time for a driver. This type of properties is a bounded *leads-to* property. An example of one such property is the following, which states that if light 1 is red then it will become green within 100 ms:

```
IF ((Aux L1) = (0,0,1)) THEN ((Aux L1) = (1,0,0)) WITHIN 100ms
```

4.7 Discussion

All these properties have been verified by the TLC model checker. In the real-time property we only used dummy values for the timing properties, but these can easily be obtained from the properties of the corresponding Hume expressions.

We can only verify relatively small models by model checking[4], and the more expressive boxes cannot be defined adequately for TLC. Furthermore, we cannot directly integrate with the work in the expression layer. Thus, we are currently working on formalising this TLA/Hume representation inside Isabelle/HOL for future integration.

5 Summary

We have presented formalisations of an assertion language for expression-level Hume in Section 3 and for coordination-level Hume in Section 4. The examples given in these sections show how to express bounded resource consumption properties on expressions and standard liveness properties on boxes.

For Hume expressions, we use a *shallow embedding of the assertion language* into the higher-order logic of the underlying theorem prover (Isabelle/HOL). In order to build up a reasoning infrastructure for proving such properties, we have developed a general VDM-style program logic for expression-level Hume. As main meta-theoretic result we have proven soundness of the program logic w.r.t. to a big-step operational semantics in Isabelle/HOL. This program logic can capture both functional properties as well as resource bounds. Therefore, it will serve as the basis for specialised logics, which can capture resource consumption in a more succinct way. Such an abstraction will be the next step in this WP, and it will enable certificate generation from high-level types, employing a multi-layered-logics approach to PCC as discussed in Section 2. Such a specialised logic will form the link between the high-level programming language and the general program logic described in this document, by mapping the meaning of high-level types, enriched with resource information, down to specialised assertions. The high-level types are exactly those used in the generic resource analysis for Hume to infer bounds on resource consumption. However, it should be emphasised that the current program logic can already be used to state resource bounds and we have given examples in Section 3.5.

On the coordination level a TLA-style logic has been formalised and used in combination with a model checker to prove liveness and safety properties of coordination-level Hume programs. In this case a *deep embedding of the assertion language* has been used. With the help of a model checker it was possible to automatically find proofs of simple TLA assertions. However, this model-checking based approach can only be used on fairly small programs, and therefore should be seen as a cheap way for automatically deriving proofs. To overcome this limitation on program size we are currently embedding the TLA formalisation into the Isabelle/HOL theorem prover. The goal is to use advanced automated theorem proving techniques, similar to those developed for the expression-level, in order to prove larger coordination-level programs. Furthermore, this effort will also enable a closer integration

[4]This state-space explosion problem is standard in model checking, and we haven't investigated how far we can push it and how to abstract to reduce the model size

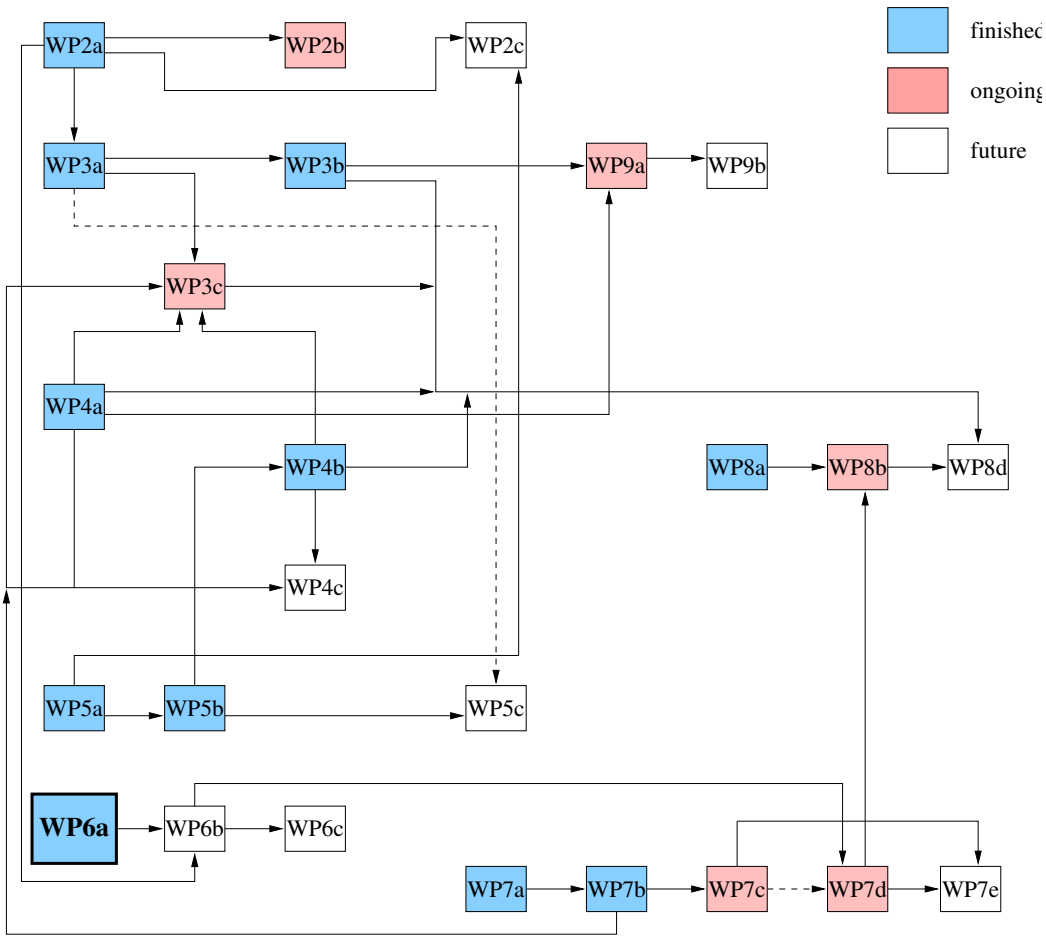

Figure 6: Project Task Dependencies

of the reasoning infrastructures on both Hume levels. Finally, by formalising our work in a state-of-the-art prover we can more easily disseminate our research results to other research groups working in this area. In particular, our multi-layered-logics approach has already attracted interest from other research groups in Madrid and Nijmegen, and we plan to cooperate more closely with these groups in the near future.

5.1 Relationship to Deliverable D26

In the work discussed in this report, we consider the verification of formal properties of Hume expressed through a novel specification notation, using a TLA embedding in Isabelle. In contrast, in D26 [Gro07] (Compiler Support Tools) we discuss the deployment of direct translation of HW-Hume into Promela for model-checking with Spin. Thus it might appear that an opportunity has been lost to develop a unified approach. However, there are important differences in requirement between this work and D26.

First of all, this report is concerned with verification of the Hume expression layer, a Turing Complete language requiring full strength semantic description and theorem proving. In contrast, D26 is concerned with the highly impoverished HW-Hume level, with minimal expression layer content. Thus, D26 is almost entirely focused on the coordination layer for which an approach intended for verifying inter-process communication and coordination properties is more appropriate. Furthermore, in D26 we are not concerned with the verification of properties requiring full strength formal proof and so the use

of a lighter-weight approach is more appropriate. It is perhaps misleading that this report discusses a model checking example in TLA.

5.2 Positioning of this Deliverable

Figure 6 gives an overview of the dependencies between workpackages, with the current WP6a highlighted. This deliverable as the result of WP6a forms the basis for the succeeding tasks in WP6b and WP6c. The structure of the logics and their relationship to one another is shown in Figure 2. The program logic, as presented in this deliverable, is the basis for the formalisation of a resource logic and for mapping down high-level types. More concretely, the next steps in WP6b will be the formalisation of a specialised resource logic, as discussed above. We will then define the format of certificates based on such a logic (D21). The final step in WP6c will be to describe and assess the generation of certificates out of high-level programs together with enriched types (D31).

References

[App01] A. W. Appel. Foundational Proof-Carrying Code. In *Proc. LICS'01 — 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE Computer Society, June 2001. 2.1, 2.2

[BJP06] F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science. Special Issue on Applied Semantics*, 364(3):273–291, 2006. 2.2

[CCNS05] B.-Y. E. Chang, A. Chlipala, G. Necula, and R. Schneck. The Open Verifier Framework for Foundational Verifiers. In *Workshop on Types in Language Design and Implementation (TLDI'05)*. ACM, January 2005. 2.2

[Gro07] G. Grov. Compiler Support Tools. EmBounded Project Deliverable, November 2007. Deliverable D26. 5.1

[HLB05] M. Hofmann, H-W. Loidl, and L. Beringer. Certification of Quantitative Properties of Programs. In *Logical Aspects of Secure Computer Systems*, Marktoberdorf, Aug 2-13, 2005. IOS Press. Lecture Notes of the Marktoberdorf Summer School 2005. 2.2

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969. 2.1

[JLH07a] S. Jost, H-W. Loidl, and K. Hammond. Report on Heap-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D11. 1, 2.2, 3.2

[JLH07b] S. Jost, H-W. Loidl, and K. Hammond. Report on Stack-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D5. 3.2

[JLH07c] S. Jost, H-W. Loidl, and K. Hammond. Report on WCET Analysis. EmBounded Project Deliverable, February 2007. Deliverable D14. 3.2

[Jon90] C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990. 3.3

[Lam94] L. Lamport. The Temporal Logic of Actions. *TOPLAS*, 16(3):872–923, 1994. 4.2

[LH06] H-W. Loidl and K. Hammond. Cost Model. EmBounded Project Deliverable, March 2006. Deliverable D4. 3.2

[Nec97] G. Necula. Proof-carrying Code. In *POPL'97 — Symposium on Principles of Programming Languages*, pages 106–116, Paris, France, January 15–17, 1997. ACM Press. 2.1

[Nec05] G.C. Necula. *Logical Aspects of Secure Computer Systems*, chapter Proof-Carrying Code. IOS Press, 2005. 2.1

[Pfe01] F. Pfenning. Logical Frameworks. In *Handbook of Automated Reasoning*, pages 1063–1147. 2001. 2.1

[SHA+05] D. Sannella, M. Hofmann, D. Aspinall, S. Gilmore, I. Stark, L. Beringer, H.-W. Loidl, K. MacKenzie, A. Momigliano, and O. Shkaravska. Mobile Resource Guarantees. In *Trends in Functional Programing*, volume 6, Tallinn, Estonia, Sep 23–24, 2005. Intellect. 2.2