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

Specific Targeted Research Project (STReP)
FET Open

D21 (WP6): Certificates

Due date of deliverable: June 2008
Actual submission date: July 2008

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

Lead contractor: Ludwig-Maximilians Universität, München Revision: 1.9

Purpose: Certificates are a representation of formal proofs of the resource properties defined in the previous stage of this WP. These certificates will be attached to the code, so that it is possible to easily check whether the stated resource properties are indeed fulfilled. These certificates should be small enough so that they can be easily attached to library code, and used in the construction of a bigger system, to ensure bounded resource consumption. The program logic itself will be based on a VDM-style, with a deep embedding of the assertion language.

Results: We have defined a concise format of certificates for Core-HUME, the internal intermediate language used in the compilation of Hume programs, in the form of proofs in a specialised resource logic. We have formalised the resource logic in Isabelle/HOL, proven soundness and given examples demonstrating that proofs in this logic are significantly shorter compared to those in the general program logic.

Conclusion: In order to facilitate the generation of certificates, we have defined, and proven sound, a resource logic, which abstracts over details of the underlying general purpose program logic for Core-HUME. We give examples demonstrating that certificates in the form of proofs of resource bounds in this resource logic are very compact.

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)	

Certificates

Hans-Wolfgang Loidl <hwloidl@tcs.ifi.lmu.de>,
Lennart Beringer <beringer@tcs.ifi.lmu.de>

Institut für Informatik, Theoretische Informatik
Ludwig-Maximilians Universität, München

Abstract

This deliverable presents a resource logic for Core-HUME, which is the main technical task in the process of designing certificates and their generation. Certificates are proofs, formalised in Isabelle/HOL, of resource assertions in this logic. We discuss meta-theoretic properties of the resource logic, in particular soundness, and give examples of proving resource assertions in this logic. The latter demonstrates, how the size and complexity of certificates can be drastically reduced, compared to proofs directly in the Core-HUME program logic.

The main work in realising our reasoning infrastructure is the soundness proof of the resource logic, which comprises more than 6,000 lines of Isabelle/HOL code but which has to be proven only once and for all. Our main design decision was to hide most of the complexity in verifying resource bounds in the rules of the resource logic.

The positioning of this deliverable is discussed in Section 7.1.

Major Revisions		
Revision	Date	Changes
1.8	14 Aug. 2009	table-of-contents (addressing Review Report Year 4)
1.5	31 July 2008	initial version

Contents

1 Introduction

In Deliverable D17 [LG07] we have defined a program logic for the Core-HUME language, which is the intermediate language used in the compilation of Hume programs. We have also defined a general format of assertions as predicates over the components of the operational semantics, using a shallow embedding into the Isabelle/HOL theorem prover. This infrastructure is already powerful enough to allow proofs of various program properties over expressions in the Core-HUME language. The formalisation of the logic in Isabelle/HOL can therefore be used to develop manual proofs of these properties. However, experience in manually developing such proofs shows that they tend to become very long, and especially the instantiation of intermediate heaps and resources, mandated by the existential quantifiers in the rules of the program logic, make it difficult to develop proofs for more complex programs.

Following our approach of multi-layered-logics, as elaborated in Section 2, we therefore develop a specialised resource logic, which is tailored for proofs of resource consumption. Central to this logic is a special form of assertions, so called "*resource assertions*", which reflect the encoding of resource consumption in the form of extended types as used in the high-level type-system (see Deliverable D11 [JLH07a]), developed for inferring resource bounds on Hume programs. By choosing such a structure we can directly take the high-level resource bounds, encode the associated types into resource assertions in this resource logic, and then generate certificates by either manually or automatically proving these resource assertions. In either case, the proof will reflect the path taken by the high-level type inference to infer the resource bounds in the first place.

In this document we focus on this *resource logic*. It has been encoded in Isabelle/HOL and soundness of the (derived) rules of this logic has been proven for the resource of heap space. Extensions to stack space and execution time are fairly simple, only requiring a modified form of resource assertions that takes these resources into account by examining the corresponding components in the resource vector. While the soundness of this logic is a crucial feature to ensure reliability of our reasoning infrastructure, the main pragmatic feature of our resource logic is the simplicity of the proofs of resource consumption in contrast to the general program logic. Whereas the latter requires instantiation of heaps and resource vectors, the former only has simple side conditions in the form of set membership and disjointness, well-typed-ness of the environment, and, of course, inequalities over natural numbers encoding the resource consumption. Since certificates are proofs in the resource logic, we thereby gain a very compact format of certificates.

We achieve such a simplicity of the proofs by a carefully chosen format of resource assertions, which hides all reasoning about complex data structures, such as the heap, in the soundness proof of the rules of the resource logic. This also means that the soundness proof of the resource logic, which has to expose this internal structure of resource assertions, becomes fairly complex. In order to achieve high reliability of our reasoning infrastructure, it is therefore important to have a machine checked version of soundness. The full soundness proof of the resource logic, including proofs of various auxiliary lemmas, encompasses more than 6,000 lines of Isabelle/HOL. In Section 3 we discuss the overall structure and the main steps of the soundness proof.

The motivation for using Core-HUME rather than HAM as language for certification and as language for the resource analyses has been given in D17 [LG07] and D11 [JLH07a] already. In short, the Core-HUME language has more structure, as is present in the high-level Hume language, and this structure helps to design program logics and type inference. One example is the presence of a conditional with two branches, rather than an (unstructured) sequence of instructions with conditional jumps in HAM code. Since the HAM code under consideration is always generated via compilation from high-level Hume code, or focus on Core-HUME as the language to build our reasoning and analysis infrastructure on is no real restriction.

The structure of this document is as follows. Section 2 gives an overview of our multi-layered-logic approach and locates the resource logic presented here in this hierarchy. Section 3 defines the notion

of a resource assertion and presents the rules of the resource logic. Section 4 discusses soundness of this logic. Section 5 presents an example of using this resource logic to prove a resource bound on a simple list-based program. Section 6 discusses related work. Section 7 concludes. Appendix A shows the Isabelle/HOL proof of the main example discussed in Section 5.

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 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 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 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

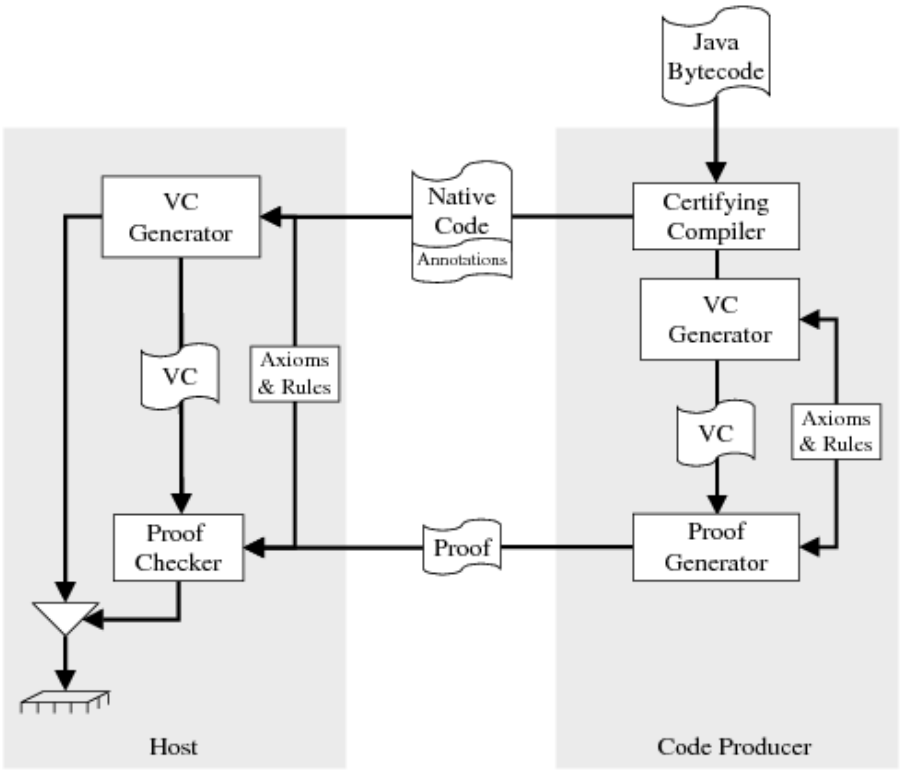

Figure 1: A classical proof-carrying code architecture (from [Nec97])

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+07, HLB05] to PCC combines advantages of both classical and foundational PCC, as discussed in the previous section. Most notably, we achieve 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+07]). 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 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 increased 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 a 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

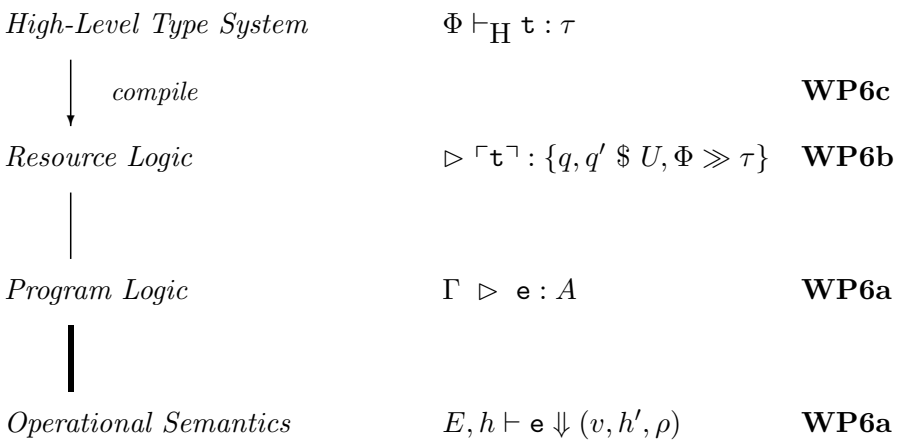

Figure 2: A family of logics for resource consumption

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 $E, h \vdash \mathtt{e} \Downarrow (v, h', \rho)$, 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. A full definition of the operational semantics for the Core-HUME language is given in Deliverable D17 [LG07]. 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 $\Gamma \ \triangleright \ \mathtt{e} : A$, where the context Γ maps expressions to assertions, and A, an assertion, is a predicate over the parameters of the operational semantics. A full definition of the program logic for the Core-HUME language, together with a formalised soundness proof, is given in Deliverable D17 [LG07]. 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 *resource logic*, which provides a reasoning infrastructure for proving bounded resource consumption in the Core-HUME language. This logic uses a restricted format of assertions, called *resource assertions*, which reflects the judgement of the high-level type system. Judgements in the specialised logic have the form $\triangleright \ulcorner \mathtt{t} \urcorner : \{q, q' \$ U, \Phi \gg \tau\}$, where the expression $\ulcorner \mathtt{t} \urcorner$ is the result of compiling a high-level term $\mathtt{t}$ down to a low-level language, and the information on resource consumption as encoded in the high-level type system is translated into a special form of assertion $\{q, q' \$ U, \Phi \gg \tau\}$, a so called *resource assertion*. 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 resource 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 $\mathtt{t}$ 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 A Resource Logic for Expression-level Hume

3.1 Source language: Core-HUME

The source language used in this formalisation is Core-HUME, as introduced in Deliverable D17 [LG07]. For convenience we repeat its definition here. 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.

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

$$\begin{aligned}\text{Expr} \ni e ::= {} & x \mid v \mid c\ x_1 \ldots x_n \mid x\ x_1 \ldots x_n \mid x_1 \oplus x_2 \mid \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \\ & \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \mathtt{case}\ x\ \mathtt{of}\ \mathit{pat}_1 \to e_1\ \mathtt{otherwise}\ e_2\end{aligned}$$

Discussion: A pattern *pat* is either a variable $x \in \text{Var}$, a value $v \in \mathcal{v}$, 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 (possibly 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 and Program Logic

The big-step operational semantics and the VDM-style program logic of Core-HUME are defined in Deliverable D17 [LG07]. Here we only summarise the relevant semantic domains and the judgement

format for the operational semantics and the program logic.

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

Semantic Domains: We use disjoint sets of identifiers for variable names ($x \in \text{Var}$), function names ($f \in \text{Funs}$), and data constructor names ($c \in \text{Constr}$). A location $l \in \text{Locn}$ is either a natural number (representing an address in the dynamic heap), or the constant nil. A heap $h \in \text{Heap}$ is a finite mapping (denoted by infix $\rightharpoonup_f$) of locations to values. An environment $E \in \text{Env}$ is a total function (denoted by infix $\Rightarrow$) of variable names to values. A semantic value ($v \in \boldsymbol{v}$) is either undefined ($\bot$), an integer value ($\mathbb{Z}$) a boolean value ($\mathbb{B}$) with elements true and false, a location (Locn), a constructor $c \in \text{Constr}$ applied to arguments at locations $l_1 \in \text{Locn} \ldots l_n \in \text{Locn}$, 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, $f \in \text{Funs}$, the number of fixed arguments $i \in \mathbb{N}$, and the values of these fixed arguments at locations $l_1 \in \text{Locn} \ldots l_n \in \text{Locn}$. We use the notation Locn^* for the domain of sequences of elements in Locn. A resource vector $p \in \text{Resources}$ 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. $|\,\text{dom}\ h\,|$. The operation freshloc has the property $\text{freshloc}\ s \notin s$, 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 $f \star \vec{xs}$ for list-map of the function f over the list $\vec{xs}$. 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 $\vec{rs}$. Similarly, we extend the update of a finite map to entire sequences like this $D(\vec{ys} \mapsto e\vec{t}ys)$.

Summary of operational semantics and program logic: In Deliverable D17 an operational semantics and a VDM-style program logic are defined. We refer to this document for details, and state here only the form of judgements. A *judgement of the operational semantics* is of the form $E, h \vdash e \Downarrow_m (v, h', p)$ and 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). In the semantics of pattern matching we use a judgement of the form $\texttt{match}\ E, h \vdash \mathit{pat}\ \texttt{at}\ l \Downarrow (E', v)$ which can be read 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 *pat* 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.

A *judgement of the program logic* for Core-HUME has the form $G \rhd e : P$, meaning that expression e fulfils 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 $P \in \text{Assn}$.

3.3 Basic Design of the Resource Logic

The basic design of the resource logic for Core-HUME is to reflect the high-level, resource-aware type system that is used on Hume level to infer upper bounds on resource consumption through annotated types. Numeric annotations are attached to type constructors and represent a "potential" that is used to "pay" for resource consumption. This basic idea is taken from amortised-cost complexity analysis [Tar85]. For example the high-level type of some list transforming function

```
0, (list[CONS<1>:int,#|NIL<1>]) -(0/0)-> list[CONS<0>:int,#|NIL<0>] ,0
```

should be read as follows. In order to succeed, this function requires 1 unit of resource for each CONS constructor and 1 unit of resource for each NIL constructor in the input list. So, if m is the length of the input list, the resource consumption of this function is $m + 1$. The annotations at the right-hand-side of the type represent resources that are free after execution of the code. For heap consumption in Core-HUME these are always 0, since heap is only reclaimed after an entire box finishes computation, i.e. outside the Core-HUME expression fragment of Hume. However, in a language that automatically reclaims heap, the standard definition of a swap function, using 1 intermediate cell, would have a 1 in the rightmost annotation of the type.

Details of this high-level type system and the semantics of these extended types are discussed in Deliverables D11 [JLH07a] and D14 [JLH07b]. Here we only recall the format of the judgement:

$$\Sigma; \Gamma \vdash^{m}_{m'} e : A \mid \Theta$$

where Γ is a type context, mapping variables to enriched Hume types, m, m' are the resource constants, e is the Hume expression, A is an enriched Hume type (for example the function type discussed above), and Θ is a set of constraints involving resource variables and constant values.

The soundness statement of this high-level type system, that is stated in these deliverables, gives rise to a definition of a resource assertion, that is tailored for reasoning about the resource consumption of Core-HUME expressions. The rules of the resource logic correspond to the rules of the high-level type system.

One immediate benefit of this design is that the inference process used to infer bounds on the high-level language can be immediately used to generate a proof of the corresponding resource assertion. Such a proof is then a *formally specified certificate of bounded resource consumption*, which can be independently verified by a proof checker that encodes the rules of the resource logic presented here. This certificate format is a very concise one, as demonstrated by example proofs in this document.

As basis for such a resource logic we define the notion of *extended types*, corresponding to high-level types.

$$\begin{array}{rcl} \mathit{EType} & ::= & \mathit{BotETy} \mid \mathit{IntETy} \mid \mathit{BoolETy} \\ & \mid & \mathit{RefETy}\ \mathit{EType} \\ & \mid & \mathit{ConETy}\ c\ \mathbb{N}^* \\ & \mid & \mathit{FunETy}\ \mathbb{N}\ \mathbb{N}\ \mathit{EType}^*\ \mathit{EType} \end{array}$$

Basic types such as *BotETy* for the singleton type of undefined values, *IntETy* for the integer type, and *BoolETy* for the boolean type are unannotated. The type *RefETy EType* represents a reference to a

heap location of type *EType*. The type *ConETy* $c\ \mathbb{N}^*$ represents a constructor (or sum) type of name c. The vector of natural numbers contains annotations (or weights) for each of the constructors of this type. Note that we model constructor types by using a table *conSigE* c which maps a constructor c to the types of its arguments and to the name of its constructor type. For example, the list type of name List with the usual constructors CONS and NIL, encoding a potential of $n+1$ (where n is the list length), would be represented as *ConETy* List $[1,1]$, with the following table entries:

$$\begin{array}{rcl} \mathit{conSigE}\ \text{NIL} & = & ([], \text{List}) \\ \mathit{conSigE}\ \text{CONS} & = & ([\mathit{IntETy}, \mathit{ConETy}\ \text{List}\ []], \text{List}) \end{array}$$

Additionally, a table *conId* is defined, which maps a constructor to its index in the list of constructors for this type. In the above example

$$\begin{array}{rcl} \mathit{conId}\ \text{LIST NIL} & = & 0 \\ \mathit{conId}\ \text{LIST CONS} & = & 1 \end{array}$$

Finally, the function type *FunETy* $\mathbb{N}\ \mathbb{N}\ \mathit{EType}^*\ \mathit{EType}$ represents a function with argument types EType^* and result type *EType*. The natural numbers encode the potential that must be available for applying the function, and the potential left over afterwards.

The resource consumption of a Core-HUME expression is encoded as the change in potential before and after execution, as illustrated in the example above. The notion of a *potential* is formally defined by induction over the underlying set of variables (the usage set). The following rules define *regions* covered by a value in a heap, encoding the region as the set of all reachable addresses, and finally the sum of all potentials. Thereby these rules establish well-formed-ness of the heap. The type of the inductively defined set is $(v \times \text{Heap} \times \mathit{EType} \times \text{Locn set} \times \mathbb{N})$ set.

$$\frac{\begin{array}{c} \#\vec{rs} = \#\vec{ety s} \\ \forall i\ .i < \#\vec{rs} \longrightarrow (\vec{rs}_i,\ h,\ \vec{ety s}_i,\ \vec{Rs}_i,\ \vec{ms}_i) \in \mathit{reg} \\ \forall i\ j.\ i < \#\vec{rs} \longrightarrow j < \#\vec{rs} \longrightarrow i \neq j \longrightarrow \vec{Rs}_i \cap \vec{Rs}_j = \emptyset \\ R = (\bigcup \mathit{set}\ \vec{Rs}) \qquad m = mf + \text{list\_sum}\ \vec{ms} \end{array}}{((f,\ ar,\ rs),\ h,\ \mathit{FunETy}\ mf\ mf'\ \vec{ety s}\ ety,\ R,\ m) \in \mathit{reg}} \quad (\textsc{regFun})$$

$$\frac{\begin{array}{c} \mathit{conSigE}\ c = (\vec{ety s},\ nam) \qquad \#\vec{rs} = \#\vec{ety s} \\ \forall i.\ i < \#\vec{rs} \longrightarrow (\vec{rs}_i,\ h,\ \vec{ety s}_i,\ \vec{Rs}_i,\ \vec{ms}_i) \in \mathit{reg} \\ \forall i\ j.\ i < \#\vec{rs} \longrightarrow j < \#\vec{rs} \longrightarrow i \neq j \longrightarrow \vec{Rs}_i \cap \vec{Rs}_j = \emptyset \\ m' = \vec{ns}_{\mathit{conId}\ nam\ c} \qquad U = \bigcup(\mathit{set}\ \vec{Rs}) \qquad m = m' + \text{list\_sum}\ \vec{ns} \end{array}}{((c,\ \vec{rs}),\ h,\ \mathit{ConETy}\ nam\ \vec{ns},\ U,\ m) \in \mathit{reg}} \quad (\textsc{regConstr})$$

$$\frac{(v,\ h,\ ety,\ U,\ m) \in \mathit{reg} \qquad h\ l = v}{((\text{Ref}\ l),\ h,\ \mathit{RefETy}\ ety,\ \{l\} \cup U,\ m) \in \mathit{reg}} \quad (\textsc{regRef})$$

$$\frac{}{(\text{Nullref},\ h,\ \mathit{BotETy},\ \emptyset,\ 0) \in \mathit{reg}} \quad (\textsc{regNullref})$$

$$\frac{}{(b,\ h,\ \mathit{BoolETy},\ \emptyset,\ 0) \in \mathit{reg}} \quad (\textsc{regBool})$$

$$\frac{}{(i,\ h,\ IntETy,\ \emptyset,\ 0) \in reg} \qquad (\text{regInt})$$

$$\frac{}{(\bot,\ h,\ BotETy,\ \emptyset,\ 0) \in reg} \qquad (\text{regUnit})$$

Note that in the regConstr rule, the regions of the components must be disjoint, and therefore sharing of data-structures is prohibited. In order to extract the weight of the current constructor c out of the list of weights attached to the constructor type in $ConETy\ nam\ \vec{ns}$, a lookup via the *conId* table is performed. The total potential is then the sum of this constructor's weight and the potential of all components. When following a reference in the regRef rule, the address is added to the region, and the potential is the same as the one for the data-structure the reference points to. The potentials of all primitive data-types is 0 and their region sets are empty.

The following rules define the *potential*, represented by a natural number, covered by a set of variables. The type of the inductively defined set is (Env × Heap × (Var *set*) × DACtxt × ℕ) *set*. A type context DACtxt is a mapping of variables to extended types, i.e. Var ⇒ *EType*.

$$\frac{U = \emptyset \qquad S = 0}{(E,\ h,\ U,\ C,\ S) \in potential} \qquad (\text{potential\_NIL})$$

$$\frac{\begin{array}{c} x \in U \qquad C\ x = ety \qquad (E\ x,\ h,\ ety,\ R,\ n) \in reg \\ (E,\ h,\ U - \{x\},\ C,\ m) \in potential \qquad S = n + m \end{array}}{(E,\ h,\ U,\ C,\ S) \in potential} \qquad (\text{potential\_CONS})$$

Informally, the potential of a usage set U of variables, is the sum of the potentials encountered in the data-structures of these variables. Note, that $E \in$ Env is a value environment, mapping variables to values, and $C \in$ DACtxt is the type enviroment mapping variables to extended types, each encoding the corresponding potential.

3.4 Resource Assertions

Central to the design of the resource logic is the format of a *resource assertion*. It encodes the information in the main soundness theorem of the high-level type system that is used to perform resource inference [JHLH09]. It is a mapping of the following judgement of the high-level type systems into an assertion of the program logic:

$$\Sigma; D \vdash^{q}_{q'} e : t \mid \Theta$$

The resource constants, q, q', the results type t and the type environment D are directly mapped to the corresponding components in the resource assertion below. The constraint set Θ in the high-level judgement restricts the value ranges for the resource variables in annotations of types in t and D, and thus defines a set of possible solutions. In the resource assertion below, we work with one concrete solution, which means the annotations in the types are always constant values, rather than resource variables, which fulfil the constraints in Θ.

The format of the resource assertion is necessarily complex, since it has to encode information about well-typed-ness, well-formed-ness of the heap, reachability of addresses, and resource bounds. The main design goal is to hide all this complexity inside the definition of a resource assertion and to prove once and for all the soundness of the rules of the resource logic.

$$
\begin{array}{l}
\{q, q' \ \$ \ U, D \gg t\} \ \equiv \ \lambda E \ h \ h' \ v \ p. \\
\quad \forall m \ r \ pot. \ okCtxt \ E \ D \ \wedge \ regionsExist \ U \ D \ h \ E \ \wedge \ regionsDistinct \ U \ D \ h \ E \ \wedge \\
\qquad (E, h, U, D, pot) \in potential \ \wedge \ m \geq q + pot + r \longrightarrow \\
\qquad (\exists R \ m' \ pot' \ F. \ (v, h', t, R, pot') \in reg \ \wedge \ Bounded \ R \ F \ U \ D \ h \ E \ \wedge \\
\qquad\quad modified \ F \ U \ D \ h \ E \ h' \ \wedge \ \mathrm{dom} \ h' = \mathrm{dom} \ h \cup F \ \wedge \ \mathrm{dom} \ h \cap F = \emptyset \ \wedge \\
\qquad\quad distinctFrom \ U \ D \ h \ E \ F \ \wedge \ m' \geq q' + pot' + r \ \wedge \ m + \mid \mathrm{dom} \ h \mid = m' + \mid \mathrm{dom} \ h' \mid)
\end{array}
$$

The arguments to the resource assertion are the "resource constants," i.e. the constant component of the available space before (q) and after (q') evaluation, the set of free variables in the expression or "usage set" U, a mapping of variables to extended types or "type environment" D, and the (extended) type of the result value t. The definition of the resource assertion is to be read like this: for a free heap space of m units and a potential *pot* units, assuming that

- the (type-)context D is well-formed w.r.t. the environment E (*okCtxt*),
- for all variables in the usage set U, type-correct regions exist in heap h (*regionsExist*),
- for all pairs of variables in the usage set U, their regions are disjoint in heap h (*regionsDistinct*),
- the total potential of all variables in the usage set U is *pot* (*potential*),
- the free heap space is initially at least $q + pot + r$;

then there exist a result region R, a fresh-set F, a result potential pot' and a free heap space m' such that

- v is the result value of type t, covering region R and containing potential pot' (*reg*)
- the locations in region R, except for the locations in the fresh-set F, are contained in the regions of the usage set U (*Bounded*),
- only locations in the regions of the usage set U have been modified between pre-heap h and post-heap h' (*modified*),
- the pre-heap h has grown by the fresh-set F, yielding a post-heap h',
- the fresh-set is disjoint from the pre-heap h,
- the fresh-set F is disjoint from the regions of the usage set U (*distinctFrom*),
- the free heap space after evaluation is at least $q' + pot' + r$,
- m is the free heap space in the pre-heap h, and m' is the free heap space in the post-heap h' (*resource*).

Note that in this resource assertion we do not refer to the resource vector p at all, since the heap size can be obtained directly from the heap, represented as a finite map. However, for other resources, such as execution time or stack space, the p component is relevant, as will be discussed in Section 3.4.1.

The auxiliary predicates, used in the resource assertion, are defined as follows. The *okCtxt* predicate states that all variables in the domain of the type environment D have the type specified in D.

$$
okCtxt \ E \ D \ \equiv \ \forall x \ ety. \ D \ x \ = \ ety \ \longrightarrow \ \mathrm{etypeof\_val} \ (E \ x) \ ety
$$

The *regionsExist* predicate states that for all variables in the usage set U, sets of well-typed addresses (Rx) and corresponding potentials (Sx) exist in the value environment E, the type environment G and heap h.

$$\begin{aligned} \mathit{regionsExist}\ U\ G\ h\ E \ \equiv\ & \forall x\ \mathit{ety}.\ x \in (\mathrm{dom}\ G) \cap U \ \longrightarrow\ G\ x = \mathit{ety} \\ & \longrightarrow\ (\exists Rx\ Sx.\ (E\ x, h, \mathit{ety}, Rx, Sx) \in \mathit{reg}) \end{aligned}$$

The *regionsDistinct* predicate states that for all pairs of variables, x, x', in the usage set U, their sets of well-typed addresses, Rx, Rx', are disjoint, in the value environment E, the type environment G and heap h.

$$\begin{aligned} \mathit{regionsDistinct}\ X\ G\ h\ E \ \equiv\ & (\forall x\ x'\ Rx\ Rx'\ Sx\ Sx'\ \mathit{etyx}\ \mathit{etyx}'. \\ & (x \in (X \cap \mathrm{dom}\ G)\ \wedge\ x' \in (X \cap \mathrm{dom}\ G)\ \wedge\ x \neq x'\ \wedge \\ & G\ x = \mathit{etyx}\ \wedge\ (E\ x, h, \mathit{etyx}, Rx, Sx) \in \mathit{reg}\ \wedge \\ & G\ x' = \mathit{etyx}'\ \wedge\ (E\ x', h, \mathit{etyx}', Rx', Sx') \in \mathit{reg}) \\ & \longrightarrow\ Rx \cap Rx' = \emptyset) \end{aligned}$$

The *modified* predicate states that the pre-heap h, and the post-heap h' differ only in addresses of regions of variables contained in Z and possibly in the addresses in F, for the value environment E, the type environment G and heap h.

$$\begin{aligned} \mathit{modified}\ F\ Z\ G\ h\ E\ h' \ \equiv\ & (\forall l.\ l \notin F\ \wedge\ l \in \mathrm{dom}\ h\ \wedge \\ & (\forall z\ Rz\ Sz\ \mathit{ety}.\ (z \in (Z \cap \mathrm{dom}\ G)\ \wedge\ G\ z = \mathit{ety}\ \wedge \\ & (E\ z, h, \mathit{ety}, Rz, Sz) \in \mathit{reg}) \\ & \longrightarrow l \notin Rz) \\ & \longrightarrow\ h\ l = h'\ l) \end{aligned}$$

The *Bounded* predicate states that all addresses in the set L are contained in regions of the variables in the usage set U, with the exception of addresses in F, for the value environment E, the type environment G and heap h.

$$\begin{aligned} \mathit{Bounded}\ L\ F\ U\ G\ h\ E \ \equiv\ & (\forall l.\ l \in L\ \longrightarrow\ l \in F\ \vee\ (\exists x\ Rx\ Sx\ \mathit{ety}.\ x \in (U \cap (\mathrm{dom}\ G)) \wedge \\ & G\ x = \mathit{ety}\ \wedge\ (E\ x, h, \mathit{ety}, Rx, Sx) \in \mathit{reg}\ \wedge\ l \in Rx)) \end{aligned}$$

The *distinctFrom* predicate states that the regions of variables in X are disjoint from the set F, for the value environment E, the type environment G and heap h.

$$\begin{aligned} \mathit{distinctFrom}\ X\ G\ h\ E\ F \ \equiv\ & (\forall x\ \mathit{ety}.\ x \in (X \cap \mathrm{dom}\ G)\ \longrightarrow\ G\ x = \mathit{ety} \\ & \longrightarrow\ (\forall Rx\ Sx.\ (E\ x, h, \mathit{ety}, Rx, Sx)\ \in \mathit{reg}\ \longrightarrow\ Rx \cap F = \emptyset)) \end{aligned}$$

3.4.1 Other Resources

Our resource logic is generic in the resource that is being modelled. The inequalities describe the generic, informal meaning that, if the concrete resources available exceed the resources required, as specified in the potential of the usage set, then the execution succeeds, with resources left-over as specified by the potential covered by the result region. Only the final clause, marked as *resource*, in the above discussion of a resource assertion, is specific to the resource under consideration. In the case for heap space consumption, this clause specifies that m is the free heap space before execution, i.e. in heap h, m' is the free heap space after execution, i.e. in heap h', and the equality expresses the fact that the amount of physical memory remains unchanged: $m+ \mid \mathrm{dom}\ h \mid = m'+ \mid \mathrm{dom}\ h' \mid$. It is easy to

adjust the resource assertion for heap space consumption, to one for time or stack space, by replacing this final clause by one that gives a different interpretation to the pre- and post-resources m and m'.

To modify the resource assertion for *execution time*, it has to refer to the clock component of the resource vector p of the assertion. In particular, the last clause has to be replaced with

$$clock\ p \leq m' - m$$

specifying that the clock at the end of the execution of the current expression has moved at most $m' - m$ ticks forward. The potential now has to be interpreted as time costs, but this doesn't change the handling of the potential in the rules. Therefore, adjusting the soundness proof to this different format of resource assertions is straightforward.

To modify the resource assertion for *stack consumption*, it has to refer to the *maxstack* component of the resource vector p of the assertion. In particular, the last clause has to be replaced with

$$maxstack\ p \leq m' - m$$

Note that the rules in the operational semantics and in the core program logic use the infix $\smile$ operator to combine resource vectors. This operator is defined to calculate the sum over the clock component, but the maximum over the *maxstack* component. Thus, the final value of this component reflects the maximal stack depth encountered during the execution. The difference in potential indicates that the available potential is at least as large as the maximal stack depth, and thereby the modified resource assertion as a whole models the consumption of a non-monotone resource.

It should be noted that the high-level type inference of heap, stack and time resources is designed in a similarly generic way. The rules of the type system are parameterised with cost constants. Supplying a table of cost constants reflecting a particular resource, instantiates the type-system to one modelling the corresponding resource consumption. We can therefore use the results of the worst-case execution time and maximal stack size analysis in the same way that we used the analysis for heap space when reading-off the resource consumption from the high level type.

3.5 Resource Logic

As mentioned before, the rules of the resource logic directly reflect the rules of the high-level type-sytem for Hume (see D11 [JLH07a]). As an example, let us examine the following rule for conditionals:

$$\frac{\Gamma \vdash^{q_1}_{q_2} e_1 : t \mid \Theta_1 \qquad \Gamma \vdash^{q_1}_{q_2} e_2 : t \mid \Theta_2}{\Gamma, x{:}\texttt{bool} \vdash^{q_1}_{q_2} \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t \mid \Theta_1 \cup \Theta_2} \qquad \text{(CONDITIONAL)}$$

It requires the variable x to have a boolean type, x:bool, and both branches must have the extended type t. The constraint sets Θ_1, Θ_2 encode bounds on the annotations in type t and other types mentioned in Γ. Matching types between both branches can be achieved by relaxing the constraints in these sets via the RELAX rule in the high-level type-system. The type environment Γ maps variables to extended types. This component is translated to the type environment D in the resource assertion. Thus, we can directly read-off the logic rule for our resource logic, by translating the high-level type judgement into a resource assertion, as shown in rule DAIF.

The rules of the resource logic are as follows.

$$\frac{\text{etypeof\_val}\ w\ t \qquad \text{basicEType}\ t}{G \rhd\ w\ :\ \{q, q\ \$\ \emptyset, D \gg t\}} \qquad \text{(DAVALUE)}$$

$$\frac{\begin{array}{c}\forall i.\ i < \#\vec{xs} \longrightarrow D\ \vec{xs}_i = \vec{ety s}_i \\ \mathit{conSigE}\ c = (\vec{ety s}, nam) \qquad \#\vec{ety s} = \#\vec{xs} \\ t = \mathit{RefETy}\ (\mathit{ConETy}\ nam\ \vec{ns}) \qquad k = \vec{ns}_{\mathit{conId}\ nam\ c} \qquad q' = q + k + 1 \\ U = \mathit{set}\ \vec{xs} \qquad \mathit{set}\ \vec{xs} \subseteq \mathrm{dom}\ D \qquad \mathrm{vars\_unique}\ \vec{xs}\end{array}}{G \rhd\ (c\ \vec{xs})\ :\ \{q', q\ \$\ U, D \gg t\}}\ (\textsc{DAConstr})$$

$$\frac{D\ x = t}{G \rhd\ x\ :\ \{q, q\ \$\ \{x\}, D \gg t\}}\ (\textsc{DAVar})$$

$$\frac{x_1 \in \mathrm{dom}\ D \qquad x_2 \in \mathrm{dom}\ D \qquad \forall v\ w.\ \mathrm{etypeof\_val}\ (v \oplus w)\ ety \wedge \mathrm{basicEType}\ \mathrm{ety}}{G \rhd x_1 \oplus x_2\ :\ \{q, q\ \$\ \{x_1, x_2\}, D \gg ety\}}\ (\textsc{DAPrimBin})$$

$$\frac{\begin{array}{c}G \rhd\ e_1\ :\ \{q_1, q_2\ \$\ U_1, D \gg t\} \qquad G \rhd\ e_2\ :\ \{q_1, q_2\ \$\ U_2, D \gg t\} \\ D\ x\ =\ \mathit{BoolETy} \qquad U = U_1 \cup U_2\end{array}}{G \rhd\ \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ :\ \{q_1, q_2\ \$\ U, D \gg t\}}\ (\textsc{DAIf})$$

$$\frac{\begin{array}{c}G \rhd\ e_1\ :\ \{q_1, q_2\ \$\ U_1, D \gg t_1\} \qquad G \rhd\ e_2\ :\ \{q_2, q_3\ \$\ U_2, D(x \mapsto t_1) \gg t_2\} \\ x \notin \mathrm{dom}\ D \qquad U_1 \cap (U_2 - x) = \emptyset \qquad U = U_1 \cup (U_2 - x)\end{array}}{G \rhd\ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\ :\ \{q_1, q_3\ \$\ U, D \gg t_2\}}\ (\textsc{DALet})$$

$$\frac{\begin{array}{c}G \rhd\ e_1\ :\ \{q_1 + k, q_2\ \$\ U_1, D' \gg t\} \qquad G \rhd\ e_2\ :\ \{q_1, q_2\ \$\ U_2, D \gg t\} \\ D\ x = \mathit{RefETy}\ (\mathit{ConETy}\ nam\ \vec{ns}) \qquad \mathit{conSigE}\ c = (\vec{ety s}, nam) \qquad k = \vec{ns}_{\mathit{conId}\ nam\ c} \\ D' = D(\vec{ys} \mapsto \vec{ety s}) \qquad \#\vec{ys} = \#\vec{ety s} \qquad x \notin \mathit{set}\ \vec{ys} \qquad \mathrm{vars\_unique}\ \vec{ys} \\ U = (U_1 - \mathit{set}\ \vec{ys}) \cup U_2 \cup x \qquad (U_1 - \mathit{set}\ \vec{ys}) \cap U_2 = \emptyset \qquad x \notin U_1\end{array}}{G \rhd\ \mathtt{case}\ x\ \mathtt{of}\ (c\ \vec{ys})\ \rightarrow\ e_1\ \mathtt{otherwise}\ e_2\ :\ \{q_1, q_2\ \$\ U, D \gg t\}}\ (\textsc{DACase})$$

$$\frac{\begin{array}{c}\{(f\ \vec{xs}, \{q_1, q_2\ \$\ U, D \gg t\})\} \cup G\ \rhd\ \mathrm{snd}\ (\mathrm{funTab}\ f)\ : \\ \lambda E\ h\ hh\ v\ p.\ \forall E'.\ E = E'(\mathrm{fst}\ (\mathrm{funTab}\ f) := E' \star \vec{xs})\ \longrightarrow \{q_1, q_2\ \$\ U, D \gg t\}\ E'\ h\ hh\ v\ (p \smile \mathcal{R}^{f\ \#\vec{xs}})\end{array}}{G\ \rhd\ f\ \vec{xs}\ :\ \{q_1, q_2\ \$\ U, D \gg t\}}\ (\textsc{DACallFun})$$

$$\frac{\begin{array}{c}\forall E\ h\ hh\ v.\ \Phi_{x,\vec{xs}}\ E\ h\ hh\ v\ x\ \vec{xs}\ \longrightarrow\ \{q_1, q_2\ \$\ U, D \gg t\}\ E\ h\ hh\ v\ \mathcal{R}^{\mathtt{ap}\ \mathrm{false}\ \#\vec{xs}} \\ (\forall f.\ \{(x\ \vec{xs}, \{q_1, q_2\ \$\ U, D \gg t\})\} \cup G\ \rhd\ \mathrm{snd}\ (\mathrm{funTab}\ f)\ : \\ \lambda E\ h\ hh\ v\ p.\ \forall E'.\ \exists \vec{rs}.\Psi_{x,\vec{xs}}\ E'\ h\ hh\ v\ f\ \vec{rs}\ \wedge\ E = E'(\mathrm{fst}\ (\mathrm{funTab}\ f) := \vec{rs} +\!\!+ (E' \star \vec{xs}))\ \longrightarrow \\ \{q_1, q_2\ \$\ U, D \gg t\}\ E'\ h\ hh\ v\ (p \smile \mathcal{R}^{\mathtt{ap}\ \mathrm{true}\ \#\vec{xs}}))\end{array}}{G\ \rhd\ x\ \vec{xs}\ :\ \{q_1, q_2\ \$\ U, D \gg t\}}\ (\textsc{DACallVar})$$

$$\frac{G\ \rhd\ e\ :\ \{q_1, q_2\ \$\ U, D \gg t\} \qquad q_1 \leq q_1' \qquad q_2' \leq q_2}{G\ \rhd\ e\ :\ \{q_1', q_2'\ \$\ U, D \gg t\}}\ (\textsc{DAWeakBnd})$$

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

Discussion: The rule for basic values DAVALUE is restricted to basic types, i.e. integers, booleans and unit, which is expressed by the basicEType predicate. The predicate etype_of w t states that the value w has the extended type t. Complex data types need to be constructed using the DACONSTR rule, so that the annotations attached to the types of the components of the constructor can be looked-up from the type environment, and thus the potential of the entire data-structure can be properly computed. Since the syntax of Core-HUME mandates let-normal-form this is no further restriction on the permitted programs. The other side-condition in the DAVALUE rule ensures well-typed-ness of the program, w.r.t. the type environment D, which is part of the resource assertion. The usage set for a constant is empty. The resource consumption of a value is 0 since the resource constants in the pre- and post-state are both q.

The side conditions in the DACONSTR rule ensure well-typed-ness, well-formed-ness, define the structure of the usage set and a non-trivial bound on the potential. The first three side-conditions ensure that all arguments to the constructor are of correct type, using the *conSigE* table to look-up the types of the arguments ($\vec{ety s}$) of constructor c and the name of type itself (nam). Since, the constructor expression represents heap allocation of a new data structure, the type of the constructor expression is a pointer to the constructor type. The potential attached to the constructor is extracted from this type by taking the ($conId\ c\ nam$)-th element of the list of weights $\vec{ns}$. The resource consumption of the constructor expression is $k + 1$, which is encoded as the difference in the resource constants q and q'. The usage set is the set of argument variables, which must be a subset of the domain of the type environment. The prefix *set* operator turns a list into a set. Finally, the vars_unique predicate states that no variable in $\vec{xs}$ occurs twice.

The rule for variables DAVAR looks-up the type for the variable in the type environment. The variable itself forms the singleton usage set. The resource consumption is 0.

The side-conditions in the DAPRIMBIN rule ensure that the expression is well-typed, w.r.t. the type environment D. The usage set is comprised of the two arguments. The resource consumption is 0, which is justified by the restriction of the primitive function to one operating over basic values.

The first non-leaf rule DAIF demonstrates how to prove resource assertions in terms of resource assertions over the components of the expression. To ensure well-typed-ness, the variable x must be of boolean type. The resource assertions for the then and for the else branch use the same resource constants, since the lookup of the value in x does not require any (space) resources, and therefore the resource consumption of the conditional is the same as those of the then and else branch. Note that the weakening rule DAWEAKBND can be used to ensure the same values, which must then be at least the maximum of both branches. The usage set of the conditional is the union of the usage sets of both branches. Note, that we do not have to add x to the usage set, since it must be of boolean type and therefore has a potential of 0.

The first two side conditions in the let rule demonstrate how the resources are threaded through the components of a let. In contrast to the conditional expression, the control-flow passes through both expressions. Therefore, an intermediate resource constant q_2 must be given, which represents the resources available after executing the let-header. The resource constant after executing the let-body is then also the resource constant at the end of the let-expression itself. The let-bound variable x must be fresh, i.e. and must not already be contained in the domain of the type environment D. As usual, the binding of x to its type t_1 is added to the type environment used for the body. The usage set for the entire let-expression is the union of the usage sets of the components, but without the let-bound variable x. Furthermore, the usage sets for the header and body must be disjoint.

The rule for case expressions covers only the non-trivial case of matching a constructor proper. The cases for matching a variable or constant are simple variants of a let binding. The side-conditions of the case rule DACase ensure well-typed-ness, well-formed-ness, define the structure of the usage set and a non-trivial bound on the potential. To ensure well-typed-ness, the type of the matching variable x must be a reference type to a constructor type of name nam. Bindings of the argument variables in the pattern to the corresponding types are added to D when proving the resource assertion for the match case. The weight of the constructor k is taken from the list of weights $\vec{ns}$ attached to the constructor type. Since the case expressions "frees" this potential, the starting resource constant in the match branch is $q_1 + k$, i.e. the additional resources encoded in the potential of the constructor c are available in this branch. In the otherwise branch the resource constants are the same as those for the entire case expression. The usage set of the case expression is the union of the usage sets of both branches, without the match-bound variables $\vec{ys}$ but with the matching variable x. The usage sets of both branches, except for the bound variables, have to be disjoint. To ensure that the potential of the constructor c is not made available several times in the program, x must not occur in the usage set of the match-branch. As trivial side-conditions of well-formed-ness, x must not occur in $\vec{ys}$ and no variable in $\vec{ys}$ is allowed to occur twice.

The rules for first-order (DACallFun) and higher-order function calls (DACallVar), are straightforward instances of the corresponding program logic rules (vdmCallFun and vdmCallVar). The reason for this simpler structure compared to the previous rules is, that we use a special form of mutual recursion rule MUTRECLemma, defined in the following section, to prove properties for an entire set of mutually recursive functions, without defining a separate logic that works over entire sets. The DACallFun rule adds the function call and the associated resource assertion to the context, and then requires a proof for a complex assertion for the body of the function. This complex assertion encodes the change in environment through parameter passing, from E to E', and applies the resource assertion to the body.

The DACallVar rule uses the same structure. As in the vdmCallVar rule, it has two side-conditions: one for an under-application, and the second for an exact application. In the former case, characterised by the predicate Φ, the resource assertion must hold for an environment E and heap h, in which the closure at variable x is an under application, specified by the $\Phi_{x,\vec{xs}}$ predicate. In the latter case, characterised by Ψ, a property has to be proven for all possible functions f, that might be encoded by this closure: with the added knowledge on the resource consumption for the function call, the given resource assertion has to be proven for the body of the function, taking into account the passing of the remaining arguments $\vec{xs}$. The value environment used in the evaluation of the body, binds the function's arguments to the values found in the closure $\vec{rs}$ and the remaining arguments to the values in the variables supplied by the function call $\vec{xs}$.

Finally, one structural rule (DAWeakBnd) is required that allows to weaken the resource constants in a resource assertion. In practice, the amount of increasing the available resource at the beginning and of decreasing the resources available after execution is often the same, and a specialised rule can be used.

3.6 Dealing with functions and mutual recursion

Standard procedure in dealing with functions and mutual recursion is to define a separate logic with judgements over entire sets of expressions, rather than just one expression. All rules, except for the function application rule, then reflect exactly the same knowledge as standard logic, which deals with just one expression. This methodology is applied on a Hoare-style logic for a simple imperative language in [Nip02].

In contrast to this standard approach, we prefer an approach that avoids the definition of such a separate logic over sets of expression. Instead, we define a predicate *goodContext*, which expresses the

fact that the assertions in a given context are strong enough to prove all entries in the context. Based on this predicate we can then define a rule that reduces the proof of a resource assertion for a function call to a proof of this *goodContext* predicate. The context required for a particular function body can be constructed by enumerating all function calls in the body and attaching to them the resource assertion derived from the high-level type of the function being called. The example in Section 5 illustrates this approach.

We first have to define the notion of *goodContext*. Its arguments are a function specification table $\mathcal{F}$, mapping a function name and an argument list to a resource assertion, and a context of expression-assertion pairs G.

$$\begin{array}{l} \mathit{goodContext}\ \mathcal{F}\ G\ \equiv \\ \quad \forall e\ P.\ (e,P) \in G\ \longrightarrow\ \exists f\ \vec{xs}.\ e = (f\ \vec{xs})\ \wedge\ (P = \mathcal{F}\ f\ \vec{xs})\ \wedge\ \forall \vec{ys}.\ G\ \rhd\ \mathrm{snd}\ (\mathrm{funTab}\ f)\ : \\ \qquad \lambda E\ h\ hh\ v\ p.\ \forall E'.\ E = E'(\mathrm{fst}\ (\mathrm{funTab}\ f) := E' \star \vec{xs})\ \longrightarrow\ (\mathcal{F}\ f\ \vec{xs})\ E'\ h\ hh\ v\ (p \smile \mathcal{R}^{f\ \#\vec{xs}}) \end{array}$$

Informally, the predicate states that all entries in the context G specify an assertion for a function call, and that this assertion can be proven for the body of the function using the same context G. In particular, this requires entries for functions encountered in the body of a function call in G. Parameter passing and modification of the resource vector directly reflect the corresponding side condition in the vdmCallFun rule.

The current version of the *goodContext* predicate is defined over the first-order fragment of the Core-HUME language. We are currently working on an extension that also covers higher-order functions, using the same approach as above with a suitably extended definition of the *goodContext* predicate.

The main auxiliary lemma used in the proof of the MUTRECLemma is a cut lemma over the context:

$$\frac{\mathit{goodContext}\ \mathcal{F}\ G \qquad (e,P) \in G \qquad D = G - \{(e,P)\}}{\mathit{goodContext}\ \mathcal{F}\ D}\ (\textsc{GoodContextCut})$$

The following rule is the general inference rule used to prove properties on a set of mutually recursive functions and is based on the definition of *goodContext* above. Note that it enables us to prove a general property P, not just a resource assertion, for a general Core-HUME expression e, not just a function call, provided this pair is an element of a finite context fulfilling the *goodContext* predicate.

$$\frac{\mathrm{finite}\ G \qquad \mathrm{card}\ G = n \qquad \mathit{goodContext}\ \mathcal{F}\ G \qquad (e,P) \in G}{\emptyset\ \rhd\ e\ :\ P}\ (\textsc{MutRecLemma})$$

Based on this general rule, we can now prove specialised rules handling function calls. The following rule is used when encountering a function application during the proof of a property of an Core-HUME expression:

$$\frac{\mathit{goodContext}\ \mathcal{F}\ G \qquad (f\ \vec{ys},\ \mathcal{F}\ f\ \vec{ys}) \in G}{(G - \{(f\ \vec{ys},\ \mathcal{F}\ f\ \vec{ys})\})\ \rhd\ f\ \vec{xs}\ :\ \mathcal{F}\ f\ \vec{xs}}\ (\textsc{AdaptCallFun})$$

Finally, the following variant is useful when proving properties of functions in the absence of mutual recursion:

$$\frac{\mathit{goodContext}\ \mathcal{F}\ \{(f\ \vec{xs},\ \mathcal{F}\ f\ \vec{xs})\}}{\emptyset\ \rhd\ f\ \vec{ys}\ :\ \mathcal{F}\ f\ \vec{ys}}\ (\textsc{EmptyProofInvs})$$

4 Soundness

The rules of the resource logic are phrased as lemmas on top of the program logic. All of these derived rules have been proven sound, together establishing soundness of the resource logic, as expressed in the following main theorem.

Theorem 1 (soundness)**.** *The rules of the resource logic in Section 3.5 are derivable in higher-order logic from the program logic specified in Deliverable D17.*

The structure of the soundness proof is as follows. After unfolding the resource assertion and applying the consequence rule, the VDM rule for the top-level Core-HUME expression is applied. Main steps in proving the unfolded resource assertion are the instantiation of the potential, of the slack space (m') and of the result region R and fresh-set F. In the non-leaf rules side-conditions over of *regionsExist*, *regionsDistinct*, *Bounded*, *modified*, and *distinctFrom*, applied to the regions of the sub-expressions have to be proven. In particular, this requires reasoning about disjointness of regions and other well-formed-ness side-conditions over the heap, as mandated by the above rules. Because of all these details the soundness proof is fairly complex: more than 6,000 lines of Isabelle/HOL[1]. The pay-off is that example proofs of resource assertions, using only these rules, are very short, as shown in the following section.

5 Examples

As example we prove a linear resource bound for a list-copy program. The definition of the code as given here is a shorthand for defining funTab copy as a pair of the argument list and the body (abbreviated as $\text{body}_{\text{copy}}$), given below .

$$\begin{array}{lcll} \text{copy}\ [x] & \equiv & \texttt{case}\ x & \\ & & (\text{CONS}\ [h,t]) \rightarrow & \texttt{let}\ y\ =\ \text{copy}\ [t]\ \texttt{in} \\ & & & \texttt{let}\ x_1\ =\ \text{CONS}\ [h,y]\ \texttt{in} \\ & & & x_1 \\ & & \texttt{otherwise} & \texttt{let}\ x_2\ = \text{NIL}\ []\ \texttt{in}\ x_2 \end{array}$$

In order to prove bounded heap consumption we use the types from the high-level resource analysis and construct the following type environment.

$$D_{\text{copy}}\ \equiv\ (x \mapsto \mathit{RefETy}\ (\mathit{ConETy}\ \text{List}\ [1,1]\ []))$$

We now prove the main theorem, expressing that the heap space consumption is bounded by $m+1$ where m is the length of the input list. The preconditions to the assertion state that variable x is a pointer to a list structure of length m in heap h inhabitating the heap locations in R. The postcondition states that the result heap is at most $m+1$ cells larger than the initial heap. Since the heap size only grows in Core-HUME programs, this establishes the desired heap bound.

$$\begin{array}{rcl} \emptyset\ \rhd\ \text{body}_{\text{copy}}\ :\ \lambda E\ h\ h'\ v\ p.\ \forall m.\ (\exists l\ v\ \mathit{ety}\ R.\ E\ x\ =\ (\text{Ref}\ l)\ \wedge\ h\ l\ =\ \text{Some}\ v\ \wedge & & \\ \mathit{ety}\ =\ \mathit{RefETy}\ (\mathit{ConETy}\ \text{List}\ [1,\ 1]\ [])\ \wedge & & \\ ((\text{Ref}\ l), h,\ \mathit{ety}, R, m) \in \mathit{reg}\ \wedge\ \text{etypeof\_val}\ (E\ x)\ \mathit{ety}) & & \\ \longrightarrow (|\ \text{dom}\ h'\ | \leq |\ \text{dom}\ h\ | + m + 1) & & \end{array}$$

The main step in the proof of the above theorem is the proof of the resource assertion below, using the rules of the resource logic. In this version the heap consumption is encoded as resource constants and

[1]This includes comments and some dead proofs.

as potential attached to the constructors of variable x in the type environment D_{copy}: a potential of 1 for the CONS constructor, and a potential of 1 for the NIL constructor. With such a potential at hand it is possible to prove the overall heap consumption of $m + 1$. The other steps in the proof of the above theorem are simple set membership and finite-map-lookup.

$$\emptyset \ \triangleright \ \text{body}_{\text{copy}} \ : \ \{1, 0 \ \$ \ \{x\}, D_{\text{copy}} \gg \mathit{RefETy} \ (\mathit{ConETy} \ \text{List} \ [0,0] \ [])\}$$

The full proof of this property is shown in Appendix A. Here we discuss only its overall structure. The proof of this property is initially syntax-directed, applying the rule of the corresponding top-level Core-HUME expression. When applying the case rule, lookups in the tables modelling the list data-type are needed. When encountering the recursive call, we apply the EmptyProofInvs rule, which relies on a proof of the *goodContext* predicate for all call sites in the code (only one in this case). Its proof is discussed below. Since the format of entries in the context is always one of a function call and an associated resource assertion, this knowledge is sufficient to prove the resource assertion for the full program.

In this example we encounter only one function call in the body of the function, and we prove the resource assertion for this one function call. The resource assertion for this recursive call uses the same type environment D_{copy} and the resource constants (2 and 1) differ by one. This encodes the same linear formula as for the overall function body.

$$\begin{array}{l} \mathit{goodContext} \ \mathcal{F} \\ \quad \{(\text{copy} \ [y], \\ \qquad \{2, 1 \ \$ \ \{y\}, D_{\text{copy}}([h, y] \mapsto [\mathit{IntETy}, \mathit{RefETy} \ (\mathit{ConETy} \ \text{List} \ [0,0] \ [])]) \gg \mathit{RefETy} \ (\mathit{ConETy} \ \text{List} \ [0,0] \ [])\})\} \end{array}$$

This property is proven by unfolding the definitions of *goodContext* and then using the rules of the resource logic with standard simplifications.

In summary we have shown how to prove bounded heap space consumption for a small, but non-trivial list-based program involving recursion. This proof uses the resource logic presented in Section 3 and does not require an unfolding of the resource assertion format. The side conditions generated by the rules of the resource logic are statements over set membership, finite-map lookup, inequalities etc, but at no point are instantiations of heap or resource vector structures required. As a consequence, this proof is much simpler than a proof in the general program logic: the main theorem is proven in 49 steps, using only basic simplification tactics, and the auxiliary lemma, proving the resource assertion is proven in 85 steps, using the resource logic rules and basic simplification rules (see Appendix A for the full proof). In contrast, direct proofs on the general program logic easily require several hundred proof steps. Another advantage of our approach of dealing with (potential) mutual recursion is the modularity of first proving resource assertions over all function calls in scope, and then proving the resource assertion of the body of the expression, using the *goodContext* predicate.

6 Related Work

This work builds directly on top of the derived assertion logic for a high-level abstraction of JVM bytecode, produced in the MRG project by Beringer et al [BHMS05]. This paper takes the same multi-layered-logics approach that we take, defines resource assertions for heap space consumption and then develops the rules of a resource logic. Additionally, this approach is used to model usage aspects [AH02], which add information about the usage of data-structures to their types. Further work in this direction is a system for layered sharing [Kon03], but this has not been formalised in such a derived assertion format. The language used in [BHMS05] is a functional abstraction of JVM byte code. In contrast to our work, it covers only first-order function calls, and is specialised for the data structures of lists and trees.

In the proof-carrying-code (PCC) community several other approaches to proving general safety conditions have been used, and we have discussed some of them in Section 2. The foundational PCC approach by Appel et al [App01] avoids the definition of a high-level type system to encode safety conditions in an attempt to minimise the size of the trusted code base. The safety conditions are encoded directly on the operational semantics instead. As a consequence the proofs of the safety properties are significantly larger than in classical PCC approaches.

Such a classical PCC approach, as advocated by Necula et al [Nec97], defines a high-level type system to encode the safety property. To validate the property typically a verification-condition-generator (VCG) together with a separate validation engine, both of which become components of the trusted-code-base, is used. One notable suggestion to minimise the size of the certificates is the use of oracle strings [Nec05]. In this approach only the decisions during high-level type inference are recorded and encoded in a so-called oracle string. The code consumer then has to replay the type inference, guided by this oracle string. Again this relies on a correct implementation of the high-level type system. In contrast, we define a general purpose logic and then prove soundness of a resource logic. This soundness proof guarantees that the proof of a resource assertion, done in the resource logic, does indeed establish a resource bound for the program.

Since our approach is one of defining a general purpose program logic, it builds on work in the program verification community. In particular, we have been influenced by the general Hoare-style program logic by Nipkow [Nip02] and by the program logic for proving correctness of pointer programs by Mehta and Nipkow [MN05]. In related work, Reynolds defines a separation logic [Rey02] to reason about disjoint areas in the heap. The predicates used in this logic have influenced our design of the predicates over regions used in the definition of resource assertions. The efforts of using separation logic to increase the level of abstraction in the verification of pointer programs are receiving much attention, in terms of foundations [O'H08], implementation [CDOY07], application for program analysis [O'H06] and formalisation in theorem provers [Web04].

7 Summary

In this document we have defined *resource assertions*, a special form of assertion of the program logic for Core-HUME, tailored to reason about resource consumption. The format of this assertion directly reflects the high-level types used in the type-based resource analysis for Hume. This close correspondence has been shown with the example of the high-level CONDITIONAL and the resource logic DAIF rules. Using these assertions we have defined a *resource logic* for proving bounded resource consumption of Core-HUME programs. This resource logic has been formalised and proven sound in Isabelle/HOL. Examples show that the proofs in this logic are significantly simpler, than proofs directly on the program logic.

The Isabelle/HOL encodings of the operational semantics, the program logic (D17) and of this resource logic (D21) form a powerful reasoning infrastructure that can be used to provide formal guarantees of both functional properties of the system as well as resource bounds. In particular, certificates of resource bounds will be Isabelle/HOL proofs using the resource logic presented here. This is a very compact format of certificates, because it uses the logic tailored for resource consumption and because it can make use of powerful Isabelle/HOL tactics to execute the proof. While being specialised to the tactic language of Isabelle/HOL, such a certificate can be checked separately from the generator of the proof, and thus no concept of trust between code generator and code consumer is required.

The examples studied so far cover the first-order fragment of Core-HUME. While the resource logic has rules for dealing with higher-order function calls, the machinery of proving sets of mutually recursive functions via the *goodContext* predicate has not been extended to higher-order functions, yet. We plan to cover this step in the final phase of Workpackage 6, when working on more substantial case

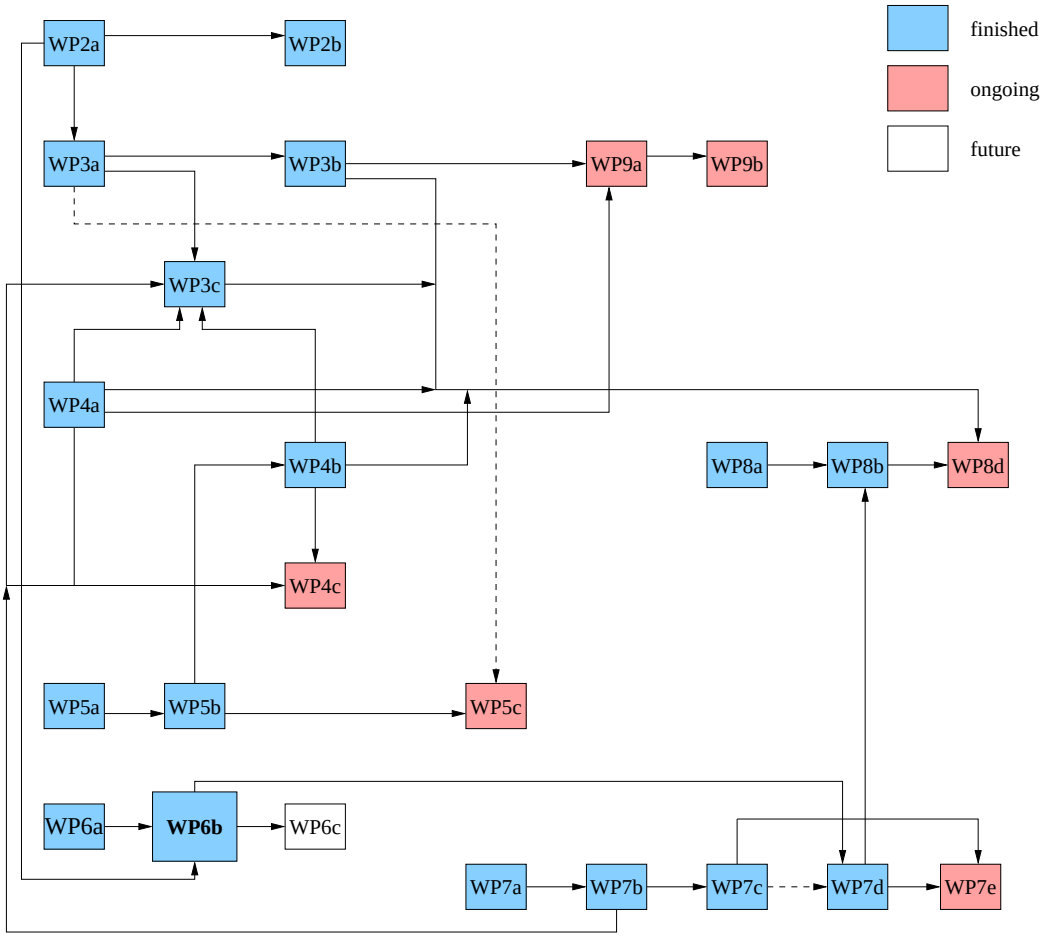

Figure 3: Project Task Dependencies

studies of certification and certificate generation. However, with the existing rules proofs of resource bounds for higher-order function calls are already possible, albeit in a more verbose format than the proof discussed in Section 5 and Appendix A, which does not need to revert to the underlying general program logic. Another deviation of the resource logic from the high-level type system is the absence of a sharing rule in the former. This means that in some cases the analysis can infer bounds, but these cannot be proven in the resource logic, because of side-conditions over the usage set. Again, it is still possible to perform the proof on the general program logic. We plan to tackle the issue of a sharing rule for the resource logic in the final phase of Workpackage 6.

7.1 Positioning of this Deliverable

Figure 3 gives an overview of the dependencies between workpackages, with the current deliverable D21 in WP6b highlighted. This deliverable builds on the operational semantics and on the program logic for Core-HUME, both encoded in Isabelle/HOL in Deliverable D17 [LG07]. The structure of the logics and their relationship to one another is shown in Figure 2. This deliverable is a prerequisite for certificate generation and for case studies on proving resource bounds in WP6c. More concretely, the next steps in WP6c will be to describe the generation of certificates out of high-level programs together with extended types (D31).

References

[AH02] D. Aspinall and M. Hofmann. Another Type System for In-Place Update. In *ESOP'02 — European Symposium on Programming*, LNCS 2305, pages 36–52. Springer-Verlag, 2002. 6

[App01] Andrew W. Appel. Foundational Proof-Carrying Code. In *LICS'01 — Symposium on Logic in Computer Science*, June 2001. 2.1, 2.2, 6

[BHMS05] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In Andrei Voronkov Franz Baader, editor, *LPAR 2004 — Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *LNCS*, pages 347–362, Montevideo, Uruguay, March 14–18, Feb 2005. Springer. 6

[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

[CDOY07] C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Footprint Analysis: A Shape Analysis that Discovers Preconditions. In *SAS07 — Symposium on Static Analysis*, LNCS 4634, pages 402–418. Springer-Verlag, 2007. 6

[HLB05] M. Hofmann, H-W. Loidl, and L. Beringer. Certification of Quantitative Properties of Programs. IOS Press, 2005. Marktoberdorf Summer School, Aug 2-13, 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

[JHLH09] S. Jost, K. Hammond, H-W. Loidl, and M. Hofmann. "Carbon Credits" for Resource-bounded Computations. *Higher-order and Symbolic Computation*, 2009. In preparation. 3.4

[JLH07a] S. Jost, H-W. Loidl, and K. Hammond. Report on Heap-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D11. 1, 2.2, 3.3, 3.5

[JLH07b] S. Jost, H-W. Loidl, and K. Hammond. Report on WCET Analysis. EmBounded Project Deliverable, February 2007. Deliverable D14. 3.3

[Kon03] Michal Konečný. Functional in-place update with layered datatype sharing. In Martin Hofmann, editor, *Proceedings of 6th International Conference on Typed Lambda Calculi and Applications*, volume 2701 of *LNCS*, pages 195–210, Heidelberg, June 2003. Springer. 6

[LG07] H-W. Loidl and G. Grov. Assertion Language. EmBounded Project Deliverable, October 2007. Deliverable D17. 1, 2.2, 3.1, 3.2, 7.1

[MN05] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.*, 199(1-2):200–227, 2005. 6

[Nec97] G.C. Necula. Proof-carrying Code. In *POPL'97 — Symposium on Principles of Programming Languages*, pages 106–116, Paris, France, Jan 15–17, 1997. 2.1, 1, 6

[Nec05] G.C. Necula. *Logical Aspects of Secure Computer Systems*, chapter Proof-Carrying Code. IOS Press, 2005. 2.1, 6

[Nip02] Tobias Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In *Computer Science Logic (CSL 2002)*, LNCS 2471, pages 103–119. Springer-Verlag, 2002. 3.6, 6

[O'H06] P.W. O'Hearn. Separation logic and program analysis. In *SAS06 — Symposium on Static Analysis*, LNCS 4134, page 181, Seoul, Korea, August 29-31, 2006. Springer-Verlag. 6

[O'H08] P.W. O'Hearn. Tutorial on separation logic. In *CAV08 — Intl Conf on Computer Aided Verification*, LNCS 5123, pages 19–21. Springer-Verlag, 2008. 6

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

[Rey02] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS'02 — Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 22–25, 2002. 6

[SHA+07] 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 (project evaluation paper). In *TFP05: Trends in Functional Programing*, pages 211–226, Tallinn, Estonia, September 2007. Intellect. 2.2

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985. 3.3

[Web04] T. Weber. Towards mechanized program verification with separation logic. In *CSL04 — Computer Science Logic*, LNCS 3210, pages 250–264, Karpacz, Poland,, September 2004. Springer-Verlag. 6

A Isabelle/HOL Proof of Resource Bounds

This appendix gives the full Isabelle/HOL proof of resource bounds for the list-copy example presented in Section 5. The main theorem is proven in 49 steps, and the lemma formalising the resource assertion is proven in 85 steps. This is in stark contrast to proofs directly on the program logic, which even for very simple programs take several hundred steps.

The following lemma proves the resource assertion, derived from the high-level type system, for the call to the list-copy function. Note that in this proof we use the rules of the resource logic (all starting with `DA`), and then do reasoning over the data-type representation of lists (rules `conSigE0..`, `conId..`). To prove the side-conditions generated by the resource logic rules, standard simplifications, reasoning over finite-map lookups, and set membership is sufficient.

```
lemma copy1DA:
  "{} \<rhd> copy0Body :
              resDA (Suc 0) 0 {xvar} copy0DACtxt (RefETy (0::nat) (ConETy List0 [0,0] [])) "
apply (simp add: funTab_copy0_def copy0Body_def)
apply (rule DACaseOnePCon8)
apply (simp add: copy0DACtxt_def)
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (simp add: conSigE0_CONS1)
apply (simp add: conId_CONS0)
apply simp+
apply (simp add: vars_unique_def)+
apply (rule DALet)
apply (subgoal_tac "{} \<rhd> CallFunExp (FN ''copy0'') [VN ''tvar''] :
                                the_funSpecTab (FN ''copy0'') [VN ''tvar'']")
  prefer 2
  apply (rule EmptyProofInvs)
  apply (subgoal_tac "goodContext0 the_funSpecTab
                       {(CallFunExp (FN ''copy0'') [VN ''tvar''],
                         the_funSpecTab (FN ''copy0'') [VN ''tvar''])}")
    prefer 2 apply (simp add: copy1good_from_copy01good  the_funSpecTab_copy1)
  apply fastsimp?
  apply (simp add: the_funSpecTab_copy1)
apply simp?
apply (rule DALet)
apply (rule DAConstr8)
defer 1
apply (simp add: conSigE0_CONS1)
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (simp add: conId_CONS0)
prefer 5
apply (rule DAVar8)
apply (simp add: dom_fmap_upds copy0DACtxt_def)
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (simp add: dom_fmap_upds copy0DACtxt_def)
apply (simp add: vars_unique_def)+
apply (simp add: dom_fmap_upds copy0DACtxt_def)
apply fastsimp
apply simp
apply (simp add: dom_fmap_upds copy0DACtxt_def)
```

```
apply fastsimp
apply simp
apply (rule DALet)
apply (rule DAConstr8)
defer 1
apply (simp add: conSigE0_NIL0)
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (simp add: conId_NIL0)
prefer 5
apply (rule DAVar8)
apply (simp add: dom_fmap_upds copy0DACtxt_def)
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (simp add: vars_unique_def)+
apply (simp add: dom_fmap_upds copy0DACtxt_def)
apply fastsimp
apply simp
apply (simp add: dom_fmap_upds copy0DACtxt_def)
apply fastsimp
apply simp
apply simp
prefer 2
apply fastsimp
apply (simp add: dom_fmap_upds copy0DACtxt_def)
apply clarsimp
apply (case_tac "i=(0::nat)")
 apply simp
 defer 1
 apply (case_tac "i=(1::nat)")
  apply simp
  apply simp
apply (rule FMAPlookup5)
apply simp+
apply (simp add: fmap_lookup_def)
apply (simp add: fifis)
done
```

The following lemma is the top-level lemma expressing a bound on heap space consumption for the example program. In particular, it states that for an input list of length m in variable x, the size of the result heap is at most $m + 1$ larger than the size of the input heap. The proof of this lemma uses the proof of the resource assertion above (`copy1DA`). After that, no reasoning about heap or resource vectors is needed — both are hidden inside the rules of the resource logic. The remaining proof steps cover reasoning over first-order logic, finite-map lookup (e.g. in `GETrSome_DOM`) to establish well-typed-ness, and set membership.

```
lemma copy1Wrapper:
 "{} \<rhd> copy0Body :
     (\<lambda> E h h' v p.
      (\<forall> m . (\<exists> l v ety R. E xvar = Rtag (Ref l) \<and>
                                           fmap_lookup h l = Some v \<and>
                                           ety = RefETy 0 (ConETy List0 [Suc 0, Suc 0] []) \<and>
                                           (Rtag (Ref l), h, ety, R, m): reg  \<and>
                                           etypeof_val (E xvar) ety)
                    \<longrightarrow> (HSize h' \<le> HSize h + (Suc m)))) "
```

```
apply (rule vdmConseq)
prefer 2
apply (rule copy1DA)
apply clarsimp
apply (simp add: resDA_def)
apply (erule_tac x="Suc m" in allE)
apply (erule_tac x="0" in allE)
apply (erule impE) apply simp
apply (rule conjI)
 apply (insert bonzo91)
 apply (simp add: copy0DACtxt_def okCtxt_def)
 apply clarsimp
 apply (subgoal_tac "x : fmap_dom copy0DACtxt")
  prefer 2
  apply (rule GETrSome_DOM)
  apply (simp add: fmap_dom_def  copy0DACtxt_def)
 apply (case_tac "x=xvar")
  apply simp
  apply (insert bonzo93)
  apply (simp add: dom_fmap_upds fmap_dom_def  copy0DACtxt_def)
apply (rule conjI)
 apply (simp add:  copy0DACtxt_def regionsExist_def)
 apply (rule_tac x=R in exI)
 apply (rule_tac x=m in exI)
 apply assumption
apply (rule conjI)
 apply (simp add:  copy0DACtxt_def regionsDistinct_def)
 apply (rule_tac x="m" in exI)
 apply (rule conjI)
 apply simp?
 apply (rule potential_CONS)
 apply (subgoal_tac "(VN ''xvar'') \<in> {VN ''xvar''}")
 apply assumption  apply simp
 apply (subgoal_tac "fmap_lookup copy0DACtxt (VN ''xvar'') =
                     Some (RefETy 0 (ConETy List0 [Suc 0, Suc 0] []))")
 apply simp
 apply (simp add: copy0DACtxt_def fmap_lookup_def)
 defer 1
 apply simp
 apply (subgoal_tac "(E, h, {VN ''xvar''} - {VN ''xvar''}, copy0DACtxt, 0) \<in> potential")
 apply simp  apply simp
 apply (rule potential_NIL)
 apply simp+  apply clarify  apply simp
 apply (insert bonzo94)
 apply (simp add: copy0DACtxt_def fmap_lookup_def)
done
```