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

Specific Targeted Research Project (STReP)
FET Open

D31 (WP6): Case studies certifying resource consumption of Hume applications

Due date of deliverable: 1st September 2008
Actual submission date: 1st September 2008

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

Lead contractor: Ludwig-Maximilians Universität, München Revision: 1.7

Purpose: In general it is unrealistic to assume that programmers will provide certificates for their code as a proof of the stated resource property. We will therefore investigate the automatic generation of such certificates, specialised for heap usage, based on extended type systems, in particular Diamond Types.

Results: We have performed case studies of verifying heap bounds on several Hume example programs, using Isabelle/HOL. The proof strategy employed there directly reflects the structure of the high-level type-inference performed by our resource analysis. Thus, these proofs demonstrate how we could generate an Isabelle/HOL proof script for each Hume function, verifying the inferred resource bound. Such a proof script then serves as certificate for bounded resource consumption.

Conclusion: We have shown how in principle certificate generation can be performed. We have not implemented a certification module in the compiler, mainly due to time constraints and shifts of effort in the project. Also, the structure of certificate generation matches the one that we have already successfully applied in the MRG project, albeit on a simpler source language.

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)	

Case studies certifying resource consumption of Hume applications

Hans-Wolfgang Loidl <hwloidl@tcs.ifi.lmu.de>

Institut für Informatik, Theoretische Informatik
Ludwig-Maximilians Universität, München
D-80339 München, Germany

Abstract

This document describes how certificates of bounded resource consumption in Core-HUME can be generated based on information from the high-level type inference on Hume. A general proof strategy for verifying resource bounds is presented, and case studies of proving resource bounds for some simple Core-HUME programs are given.

A discussion of how the success criteria have been met is given in Section 4.1. The positioning of this deliverable is discussed in Section 4.2.

Major Revisions		
Revision	Date	Changes
1.6	14 Aug. 2009	table-of-contents, positioning in Sec 4.2 (addressing Review Report Year 4)
1.5	1 Sep. 2008	initial version

Contents

1 Introduction

In Deliverable D21 [LB08] of this workpackage we have defined certificates as proofs of resource assertions in a specialised resource logic. We have shown that such certificates are fairly compact because the resource logic is tailored to reflect the structure of the type-inference-based resource analysis of Hume programs. In this document we discuss how this close match between high-level type inference and the resource logic can be exploited to automatically generate certificates.

We follow an approach of multi-layered-logics, as elaborated in Deliverable D21 [LB08]. As foundation of this hierarchy of logics we have defined a general program logic for the Core-HUME language, which is the intermediate language in the compilation of Hume programs. Soundness of this program logic has been established w.r.t. a resource-aware operational semantics, and all of this is formalised in Isabelle/HOL. We have then developed a specialised resource logic, which is tailored for proofs of resource consumption. The assertions in this logic, so called "resource assertions", reflect the encoding of resource consumption in the form of extended types, a generalisation of Diamond Types [Hof00], as used in the high-level type-system (see Deliverable D11 [JLH07]), 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. A direct comparison of a proof with an inference path is given in Appendix A.1. The difficult task of certificate generation, i.e. of finding a proof for a program property, is therefore reduced to the fairly mechanical task of recording the decisions made when constructing an extended typing for the Hume program.

In the remainder of this document we first describe such a process of certificate generation in general in Section 2, and in particular a proof strategy for verifying resource assertions. We then discuss in Section 3 several case studies of manually verifying resource assertions, following this proof strategy, and we discuss possibilities of automating this process. In Section 4 we summarise the status of our verification infrastructure. The appendices to this document contain complete Isabelle/HOL proof scripts of the verifications in Section 3.

2 Certificate Generation

2.1 Resource Logic

We shortly summarise the essence of our resource logic as developed in Deliverable D21 [LB08]. The resource logic builds on the general, VDM-style program logic for Core-HUME as developed in Deliverable D17 [LG07]. 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 $P \in \text{Assn}$. Notably, an assertion can be any predicate over the components of the operational semantics. Thus, assertion can make statements over the functional correctness of the program, over the resource consumption, or any other aspect of the program execution made explicit in the underlying operational semantics.

In contrast, a *judgement of the resource logic* for Core-HUME has the form $G \rhd e : \{q, q' \$ U, D \gg t\}$. The format of the assertion is restricted to a form that reflects the high-level typing produced by our high-level type inference. More concretely, the definition of the resource assertion is a formalisation of the main soundness theorem of the high-level type system. 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*).

In particular, the heap space after execution is bounded by the value, $q' + pot' + r$, which is calculated from the resource constant q', the potential pot' of used variables, and some slack space r. As a concrete 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.

The certificate of bounded resource consumption for this program consists of the following judgement $\{0, 0\ \$\ \emptyset(x \mapsto \mathit{ConETy}\ \text{ListTy}\ [1,1]), \{x\} \gg \mathit{ConETy}\ \text{ListTy}\ [0,0]\}$ and the corresponding proof in the resource logic, as will be discussed in detail in the case studies in Section 3. Here, the type-context binds the argument of the function to a list type with potential 1 attached to both the CONS and NIL constructors, and the result type is a list with weights 0. The proof of this resource bound uses only rules of the resource logic, which are syntax-directed except for a weakening rule that allows to weaken the resource bounds. The following section discusses the proof strategy in more detail and discusses how such a proof can be automated. After that, we present several case studies of manually proving resource bounds.

2.2 Proof Strategy

The basic proof strategy is simple since most of the rules of the resource logic are syntax-directed. The weakening rule is applied at the beginning of a function body and in the conditional and case rules, to have the same bounds for both branches. The open issues are the treatment of, possibly recursive, functions and the simplifications needed in-between applying the rules of the resource logic.

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 the 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 the 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 from static information about the program, as described below. The examples in Section 3 illustrate this approach.

We first recall the definition of *goodContext* from Deliverable D21. 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 in the program logic.

Based on this definition we can now define the following rule for proving an assertion stored in the function specification table $\mathcal{F}$ for a function call. This rule has been proven sound in the Isabelle/HOL formalisation. Note, that the argument lists in the context G and at the concrete function call may be different, but they have to correspond when performing a lookup in the $\mathcal{F}$ table. In other words, this rule has adaptation, or renaming, of function arguments built-in.

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

A slight variant of the AdaptCallFun rule, is the rule CallFunByGoodCtxt below. It has the advantage that we can apply it with an empty context. Again it is based on the (finite) context G fulfilling the *goodContext* predicate. It reduces the proof of a function call $f\ \vec{xs}$ to a lookup into the context G for a corresponding argument list $\vec{ys}$. As in the rule above, the entries in the context are parametric over their arguments, allowing for adaptation, or renaming, of the arguments. The CallFunByGoodCtxt is proven, again in Isabelle/HOL, by first applying EmptyProofInvs, which in turn is proven by applying AdaptCallFun, and then performing induction over the size of the set G. In the induction step, a standard cut rule on the *goodContext* predicate is applied.

$$\frac{\mathit{goodContext}\ \mathcal{F}\ G \qquad \text{finite}\ G \qquad (f\ \vec{ys}, \mathcal{F}\ f\ \vec{ys}) \in G}{\emptyset\ \rhd\ f\ \vec{xs}\ :\ \mathcal{F}\ f\ \vec{xs}}$$

(CallFunByGoodCtxt)

To summarise, the proof strategy for a function body is as follows:

- determine all function calls in the call graph starting from the function body;
- construct a context, consisting of these calls and attaching to each call the resource assertion that can be read-off the high-level type for this function;
- prove the *goodContext* predicate for this context; this amounts to proving a resource assertion for the body of the encountered function, which uses the same proof strategy;
- prove the body of the function, using the AdaptCallFun or CallFunByGoodCtxt rule.

The list-copy and control code case studies in Section 3 demonstrate this proof strategy, on simple, recursive programs.

Constructing a good context: Our proof strategy relies on a context that is strong enough to prove all resource assertions for function calls encountered when proving a function body. The construction of such a context is mechanical and only requires static information, which is inferred by the compiler during type- and resource inference. Each entry in the context is an expression-assertion pair. The expression is always a function call, and the assertion is a lookup in the function specification table $\mathcal{F}$, which contains only resource assertions. Thus, the context for proving a function body is built from

- the information on the call-site (function name and argument names),
- the type information (including annotations) on the arguments, and
- the type information on the result of the function.

Simplifications: Examination of the rules of the resource logic shows, that most of the side-conditions generated are simple and do not need sophisticated proof search to prove: we find statements over set membership, finite-map lookup, inequalities etc. Consequently, the simplifier used in proving a resource assertion builds on the standard simplifier in Isabelle/HOL, adding only a few lemmas on finite maps and lookups on the tables defining the data types. The case studies in the following section elaborate on this issue.

3 Case Studies

3.1 Case study: Pair-swap

To illustrate the proof strategy described in Section 2.2 we start with a deliberately simple example: swapping two elements in a pair.

The definition of the code as given here is a shorthand for defining "funTab swap" as a pair of the argument list and the body (abbreviated as $\text{body}_{\text{swap}}$), given below.

$$\begin{array}{lcl} \text{swap } [x] & \equiv & \texttt{case}\ x \\ & & \quad (\text{PAIR } [\mathit{left}, \mathit{right}]) \rightarrow \quad \text{PAIR } [\mathit{right}, \mathit{left}] \\ & & \quad \texttt{otherwise} \quad x \end{array}$$

In the initial type context we bind the argument of the function x to a pair type with a potential of 0:

$$D_{\text{swap}} \equiv \emptyset(x \mapsto \mathit{ConETy}\ \text{PairTy}\ [0]\ [])$$

The following resource assertion formalises a constant heap consumption of 1 cell, via the first resource constant, and no heap left-over after execution, via the 0 potential in the result type of the resource assertion.

```
lemma "{} \<rhd> swapPairBody : resDA 1 0 {xvar} swapPairDACtxt (RefETy (0::nat) (aPairETy 0 ))"
apply (simp add:  swapPairBody_def)
apply (rule DACaseOnePCon8)
apply (simp add: swapPairDACtxt_def)
apply (simp add: aPairETy_def)
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (simp add: conSigE0_PAIR)
apply (simp add: conId_PAIR)
apply simp
apply simp
apply simp
defer 1 (* vars_unique *)
apply (rule DAConstr8)
defer 1
apply (simp add: conSigE0_PAIR)
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (simp add: aPairETy_def)
apply (rule conjI) apply simp+
apply (simp add: conId_PAIR)
apply simp+
apply (rule conjI)
apply (simp add: dom_fmap_upds)+
defer 1 (* vars_unique *)
apply (rule DAWeak_Bnd)
apply (rule DAVar8)
apply (simp add: swapPairDACtxt_def)
apply (subgoal_tac "Suc 0 \<le> Suc 0")
apply assumption
apply simp+
defer 1 (* vars_unique *)
apply (simp add: swapPairDACtxt_def)
```

```
apply (subgoal_tac "VN ''xvar'' \<in> {VN ''xvar''}")
(* match types *)
apply clarify?
apply (case_tac "i=0")
 apply clarsimp
 defer 1 (* ok *)
 apply (case_tac "i=1")
  apply clarsimp
  defer 1 (* ok *)
  apply clarsimp
 apply simp
defer 1 (* vars_unique *)
defer 1 (* vars_unique *)
 apply (simp add: fmap_lookup_def)
 apply (simp only: fifis)
 apply simp
 apply (simp add: fmap_lookup_def)
 apply (simp only: fifis)
 apply simp
(* unfold vars_unique *)
 apply (simp add: vars_unique_def)+
done
```

We now examine the structure of the proof. Notably, the selection of rules from the resource logic is mostly syntax-directed: after unfolding the body of the function, we use the `DACaseOnePCon8`, or DACase rule[1] on a top-level case construct, then the `DAConstr8`, or DAConstr, rule followed by the `DAVar8`, or DAVar, rule on the branches of the case-expression. In order to match the resource bounds of both branches, the weakening rule `DAWeak_Bnd`, or DAWeak, is used. Inbetween and after applying these rules of the resource logic, naturally, some simplification is necessary. However, at no point are complex proof searches needed to discharge the side-conditions left-over by the rules of the resource logic. Most of the side-conditions after the `DACaseOnePCon8`, or DACase, rule are predicates of set membership and can be solved by the standard simplifier. Since the case-expression frees the potential attached to the constructor, we need a table lookup into `conSigE0_PAIR` and `conId_PAIR` to extract the argument types and the potential of the `PAIR` constructor, respectively. The generated inequalities are trivial and do not require a special arithmetic tactic to prove; again standard simplification is sufficient. The side-conditions over the type-context, generated by `DACaseOnePCon8` and `DAConstr8`, are solved by using a small set of lemmas over finite maps and updates over finite maps, e.g. `dom_fmap_upds`. The last 19 steps of the proof establish that the `PAIR` constructor is applied to arguments of the correct type, and again these need lemmas over finite maps[2]. In total, this resource assertion was proven in 58 steps.

In order to automate the proof discussed above, a simplifier, tailored for proofs of these resource assertions should be defined. This simplifier should extend the standard one, with rules for table lookup for all user defined constructors `C`: `conSigE0_C` and `conId_C`. Furthermore, the expected lemmas over finite maps w.r.t. lookups, construction of domain etc in the presence of updates need to be added. We do not expect to need a general arithmetic simplifier, since the generated inequalities have a very simple form and should be solved on-the-fly with such an extended simplifier. This corresponds to our experience in the MRG project, where we defined an Isabelle/HOL strategy, capable of proving bounded heap consumption in the format discussed here. In this strategy, it was sufficient to add the

[1] In some cases the names of the rules in the formalisation differ from those specified in D21. Where necessary we give both names.

[2] All our formalisations are done in Isabelle2005, with a finite-map theory we defined for our purpose. More recent versions of Isabelle have pre-defined finite-map theories, which could reduce the number of proof steps even further.

rules described above in order to prove a range of high-level programs.

3.2 Case study: List-copy

We now examine a simple recursive program, copying a list, in order to demonstrate how our proof strategy works in the presence of recursion. This example has first been used in Deliverable D21, and the proof of the resource bound has been given there, too. In this document we focus on the possibility of generating a proof script as a certificate of bounded resource consumption.

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.

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

Following the proof strategy of Section 2.2, we prove that the resource assertion, constructed from the high-level type, holds for the body of the function.

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

The full proof of this property is shown in Appendix A.1. 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 definition of *goodContext* and then using the rules of the resource logic with standard simplifications.

Appendix A.1 also shows a trace of the rules of the high-level type-system, encountered when inferring heap consumption. A comparison with the structure of the proof discussed here, and shown in detail in Appendix A.1, demonstrates that we can, in principle, generate a certificate directly out of such a trace, when we add appropriate calls to the simplifier in-between the calls to the rules of the resource logic (which correspond to the rules of the high-level type-system). There are still some design decisions to be taken, in particular whether to apply simplification on-the-fly or to first generate all

side-conditions and then do one, or several, simplification steps. The latter approach is more common in the theorem proving community, by constructing a verification-condition-generator as a standalone tool, which feeds its results into a powerful simplifier. However, based on our experience with certificate generation and automatic validation in the MRG project, we prefer an approach of doing simplifications on-the-fly, ie. in-between the rules of the resource logic. This keeps the set of remaining side-conditions small, which helps the standard simplifier to solve this set without external help.

3.3 Case-study: Control code

Our last case study is taken from the CyCab control code. The application is described in Deliverable D27 and its role in the EmBounded project in general is assessed in Deliverable D33. In this document we are mainly interested in the certification of heap bounds. Therefore, we phrase the heap consumption of the code in the form of a resource assertion, prove this assertion within our resource logic in Isabelle/HOL, and comment on the potential for automating this process.

Due to time constraints and the lack of automation of our proof strategy, we restrict the study to one branch in the `robucar _in` control box of the CyCab control code. This box is in charge of transfering data between the control component of the CyCab and the lane-tracking box that performs the brunt of the image processing. The operations performed in this branch are therefore fairly simple, but characteristic for the applications encoded within one Hume box.

As before, the definition of the code as given here is a shorthand for defining "funTab control" as a pair of the argument list and the body (abbreviated as $\text{body}_{\text{control}}$), given below.

```
control [x]  ≡  case x
                  (TIN [w, v, n, enid, enid0, size, crc]) →
                   let q1 = (λx y. x == 2) n n in
                   if q1
                   then  let k2 = 2 in
                         let x1 = update [v, k2, w] in
                         let crc' =  let q2 = (λx y. x == 9) enid enid in
                                     if q2
                                     then wvtow16le [x1]
                                     else wvtow16be [x1]
                         let k1 = 1 in
                         TOUT [z, k1, enid0, size, crc']
                   else TOUTNOT []
                  otherwise  TOUTNOT []
```

This box-level code reads a seven-tuple as input, representing the current state of the low-level control component of the CyCab. It performs some simple computation and data-type conversion in its main branch. First, well-formedness of the communication protocol is checked by examining the value of n. In encoding primitive operations we use the usual λ notation. Thus, q_1 represents the boolean value of the variable n containing the value 2. If this is successful, the main input data-structure, v, is updated at position 2 with the value w, and the current checksum is modified from *crc* to crc', reflecting the current communication action. In performing this modification, 2 low-level data conversion functions, wvtow16le and wvtow16be, are used, depending on whether the protocol uses little- or big-endian representations, which is encoded in the *enid* variable of the input. The result of the main branch is the TOUT constructor applied to five arguments: the global constant z, the local variable k_1, the endian-ness information in *enid0*, the size in *size* and the new checksum value in crc'. In the case, where the message doesn't comply with the communication protocol, ie. if the test of n for 2 is false, an empty result is produced, represented as an TOUTNOT constructor. The same result

is produced if an input other than a seven-tuple is read, which also covers the case of no input on box-level. It would be easy to use different constructors here, and the heap consumption would remain the same.

We now define the context needed to prove a resource assertion for this code. The body of the control function contains three function calls, so we need three entries in the context, each attaching a resource assertion to the corresponding function call. The update function updates a list at a given position with a given value. Here we assume a heap consumption of 1 heap cell, ie. performing an in-place update. To change this to a model of a non-destructive update, only the type of the v variable has to be changed to $\mathit{RefETy}\ (\mathit{ConETy}\ \text{List}\ [1,1]\ [])$, as we have shown already in the list-copy example. The functions wvtow16le and wvtow16be are low level data-type conversions on words, which are modelled as *IntETy* types in our formalisation, and we assume 0 heap consumption for these two operations. The former covers the case of little endian-ness, the latter of big endian-ness.

The type context used in all three cases contains the arguments to the function, and additionally all arguments for the control function. The latter is not strictly necessary, but it saves the application of one consequence rule. Most importantly, all entries to the context can be generated from the static information available to the compiler, as discussed in Section 2.2:

- the information on the call-site (function name and argument names),
- the type information (including annotations) on the arguments, and
- the type information on the result of the function.

$$
\begin{array}{l}
\mathit{goodContext}\ \mathcal{F} \\
\quad \{(\text{update}\ [v, k_2, w], \\
\qquad \{1, 0\ \$\ \{v, k_2, w\}, \\
\qquad\quad (D_{\text{control}} \\
\qquad\qquad [w, v, n, \mathit{enid}, \mathit{enid0}, \mathit{size}, \mathit{crc}] \\
\qquad\qquad [\mathit{IntETy}, \mathit{RefETy}\ (\mathit{ConETy}\text{List}\ [0,0]\ []), \mathit{IntETy}, \mathit{IntETy}, \mathit{IntETy}, \mathit{IntETy}, \mathit{IntETy}]) \\
\qquad \gg \mathit{RefETy}\ (\mathit{ConETy}\ \text{List}\ [0,0]\ [])\}), \\
\quad (\text{wvtow16le}\ [x_1], \\
\qquad \{0, 0\ \$\ \{x_1\}, \\
\qquad\quad (D_{\text{control}} \\
\qquad\qquad [w, v, n, \mathit{enid}, \mathit{enid0}, \mathit{size}, \mathit{crc}] \\
\qquad\qquad [\mathit{IntETy}, \mathit{RefETy}\ (\mathit{ConETy}\ \text{List}\ [0,0]\ []), \mathit{IntETy}, \mathit{IntETy}, \mathit{IntETy}, \mathit{IntETy}, \mathit{IntETy}]) \\
\qquad \gg \mathit{IntETy}\}), \\
\quad (\text{wvtow16be}\ [x_1], \\
\qquad \{0, 0\ \$\ \{x_1\}, \\
\qquad\quad (D_{\text{control}} \\
\qquad\qquad [w, v, n, \mathit{enid}, \mathit{enid0}, \mathit{size}, \mathit{crc}] \\
\qquad\qquad [\mathit{IntETy}, \mathit{RefETy}\ (\mathit{ConETy}\ \text{List}\ [0,0]\ []), \mathit{IntETy}, \mathit{IntETy}, \mathit{IntETy}, \mathit{IntETy}, \mathit{IntETy}]) \\
\qquad \gg \mathit{IntETy}\})\}
\end{array}
$$

The initial type context, D_{control}, used in the context above, contains bindings for the arguments, x, of the function and for the constants, z, used in it:

$$D_{\text{control}} \ \equiv\ ([x, z] \mapsto [\mathit{RefETy}\ (\mathit{ConETy}\ \text{InTy}\ [0,0]\ []), \mathit{IntETy}])$$

Now, the top-level judgement to be proven specifies a resource consumption of 2 heap cells for the body of control, producing a value of type OutTy as result:

$$\emptyset\ \rhd \text{body}_{\text{control}}\ :\ \{2, 0\ \$\ \{x, z\}, D_{\text{control}} \gg \mathit{RefETy}\ (\mathit{ConETy}\ \text{OutTy}\ [0,0]\ [])\}$$

```
lemma robucar5DA: "{} \<rhd> robucar5Body :
                              resDA (Suc (Suc 0)) 0 {xvar,zconst} robucar5DACtxt
                                    (RefETy (0::nat) (ConETy OutTy [0,0] []))"
apply (simp add: robucar5Body_def)
apply (rule DACaseOnePCon8)
apply (simp add: fmapLookup_aux)
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (simp add: conSigE0_TIN)
apply (simp add: conId_TIN)
apply simp+
apply (simp add: vars_unique_def)+
apply (rule DALet)
apply (rule DAPrimBin)
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply (simp only: equals_2_def)
apply (simp add: basicEType_aux)
apply (rule DAIf)
apply simp
apply (rule DALet)
apply (rule DAValue)
apply (simp add: etypeof_val_def basicEType_def)
apply (simp add: etypeof_val_def basicEType_def)
apply (rule DALet)
```

Figure 1: Initial syntax-directed part of the proof

We now discuss the structure of the proof in more detail. For reference, Appendix A.2 shows the complete proof.

The first part of the proof, shown in Figure 1, is again syntax-directed. The side-conditions of the case rule `DACaseOnePCon8`, or DACase, are solved by lemmas on finite-map lookups, data-type table lookup via the `conSigE0..` and `conId..` tables, and by standard simplifications (`simp+`). The `vars_unique` predicate, stating that every variable occurs only once the argument list, is also solved by unfolding the definition and standard simplifications. The side-conditions of the `DAPrimBin`, or DAPrimBin, rule, in the header of the first let, only have to establish well-typed-ness of the primitive operation, which is again done by finite-map lookups on the variables in the type context. The `DAIf`, or DAIf, rule checks well-typed-ness of the header variable, and leaves resource assertions on the then- and else-branch as top-level subgoals. The then-branch again contains a let construct with a constant being assigned in the header, thus `DALet`, or DALet, followed by `DAValue`, or DAValue, are used. In the next let-header the update function is called.

Figure 2 shows the proof steps for the update function. For all three instances of function calls we use the same sequence of proof steps. The rule that we use to prove the function call is CallFunByGoodCtxt, as discussed in Section 2.2. Based on this rule, the function call can be proven by using the proof of *goodContext* above, followed by standard simplifications. The initial subgoal tactic, used at the beginning of Figure 2 is easy to automatically generate: it is a judgement of the call-site, with a table lookup to the function specification table $\mathcal{F}$ applied to the arguments in this call-site. After the proof of the resource assertion for the function call, the weakening rule is applied to the resource assertion found in the context. This could be avoided by having several entries with tailored resource consumption in the context. We prefer this version, which achieves better modularity of our proof strategy, in the form of

```
(* CallFunExp updateFun ... *)
apply (subgoal_tac "{} \<rhd> CallFunExp (FN ''updateFun'') [VN ''vvar'', VN ''k2var'', VN ''wvar''] :
                                the_funSpecTab (FN ''updateFun'') [VN ''vvar'', VN ''k2var'', VN ''wvar'']"
  prefer 2
  apply (rule CallFunByGoodCtxt)
  apply (insert robucar5good) apply simp
  apply (simp add: robucar5Ctxt_def)
  apply (simp add: robucar5Ctxt_def)
  apply (rule conjI) apply simp
  apply simp
  apply (simp add: the_funSpecTab_updateFun)
(* done with CallFunExp *)
apply (rule DAWeak_Bndk)
apply fastsimp?
apply simp+
```

Figure 2: Proof of function calls

smaller contexts, at the expense of an additional application of the `DAWeak_Bndk` rule per function call.

The remainder of the proof script follows the same structure and is only shortly summarised. The next expressions are two nested let expressions, with a conditional in the inner let. Each branch of the conditional is a function call, which is solved using the same sequence of proof steps as above, including weakening. The final let header binds a constant, which is solved by applying `DAValue` and proving well-typed-ness. The body of the nested let is a constructor application of TOUT with five arguemnts. This requires an application of the `DAConstr8` rule, followed by lookups of the data-type tables, and checking well-typed-ness through finite-map lookups.

The remaining branches, the else-branch of the inner conditional and the otherwise-branch of the case, are both constructor applications of TOUTNOT. In order to make the resources of all branches match-up, first a weakening rule is applied, then the `DAConstr8`, or DAConstr, rule is applied, again followed by lookups of the data-type tables, and checking well-typed-ness through finite-map lookups. This completes the syntax-directed part of the proofs.

In the remaining sub-goals, inequalities over meta-variables, representing resources are contained. These inequalities always have a simple structure, since they are generated from applications of the `DAWeak_Bndk` rule, over resource constants and weights attached to constructors, which are in turn simplified in each step. Since the resource constants and weights are provided as constants by the high-level type system, which has obtained the concrete values by applying a linear programming approach, the search space in the prover when validating the bounds is limited and shouldn't be difficult to automate. In this proof, however, we explicitly instantiate inequalities, using a subgoal-tactic of $1 \leq 1$, to help the simplifier. The remaining steps in the proof are mainly tests for set membership.

In summary, we have proven bounded heap consumption for one branch of a box, used in the CyCab application. Despite the example itself being fairly simple, it is realistic, and it demonstrates how a certificate in the form of an Isabelle/HOL proof script can be generated, largely following the proof strategy outlined in Section 2.2. There are still some decisions open before fully automating this strategy, eg. by defining an Isabelle tactic. One decision is when to apply the weakening rule. It is needed to match the resources in branches of conditionals and case expressions. We have also applied it together with a variant of the function application rule, CallFunByGoodCtxt, which is powerful enough to handle mutually recursive sets of functions. Another decision is when exactly to apply simplification on inequalities. Our experience shows that simplification on-the-fly keeps the inequalities simple and avoids the usage of a powerful arithmetic simplifier. In our example we helped the standard simplifier at two points to instantiate the right resource constants.

4 Summary

In this document we have discussed how certificates of bounded resource consumption of Core-HUME programs can be generated out of high-level types as inferred by our resource analysis. In our system, certificates are formal proofs of resource assertions in a special logic, a resource logic as defined in Deliverable D21. The rules in this resource logic are largely syntax-directed. Therefore, in most cases the choice of the next rule only depends on the top-level program construct. The only open issues are when to apply weakening, how to deal with mutually recursive functions and which simplifications to perform. We have addressed all of these issues in Section 2.2. Weakening is best performed on the branches of conditionals and case expressions. For handling mutual recursion we define a new predicate *goodContext*. With a mechanically constructed context, fulfilling this predicate, proving a resource assertion on a function call can be reduced to proving that an adapted expression-assertion pair is an element of such a context. This approach delivers a modular way of proving properties on a program with a set of mutually recursive functions. Finally, we examined the simplifications necessary in our examples. We found that the standard Isabelle/HOL simplifier, extended with only a few lemmas on our finite-map theory and on table lookups, was sufficient to prove the given resource assertions. Where inequalities could not be eliminated directly by the simplifier, a search over a narrow set of natural numbers was sufficient to prove the remaining inequalities.

Although we haven't implemented this proof strategy as a tactic in Isabelle/HOL, we are confident that it forms the right basis for proving a given resource assertion. Thus, certificate generation amounts to constructing lemmas of resource bounds, which are proven by an application of this proof strategy. For each function exactly one lemma needs to be generated, where the resource assertion can be directly constructed out of the high-level type. In the presence of mutual recursion, a context with all call sites, and the corresponding resource assertions, has to be constructed and the *goodContext* predicate has to be proven for this context. In a previous project (MRG), we implemented such a proof strategy for a simpler language and were able to verify bounded heap space consumption for a range of programs.

The case studies presented in this document use simple example programs to study specific language features. Due to time constraints in the project and the lack of automation of the proof strategy, we have used only 2 simple example programs and one branch in a box of a Hume control application from Workpackage WP8. However, these examples demonstrate how to handle recursion, compound data-types and discuss which simplification steps are necessary to complete the proof.

The most serious shortcoming of our current infrastructure is the absence of a sharing rule in the resource logic, which prohibits repeated use of variables. In such cases, it is no longer possible to only use the rules of the resource logic. Instead, the rules of the underlying program logic, with their more complicated side-conditions have to be applied. In the high-level type system, repeated use is permitted, provided that the sum of the potentials of all uses doesn't exceed the total potential of the variable that is used multiple times. We haven't yet proven a corresponding rule in the resource logic, mainly due to the technical overhead of the soundness proof rather than some conceptual problems.

4.1 Meeting the Success Criteria

The success criterion for this task as stated in the Requirements Analysis (D2) is:

> Success criteria: A description of certificate generation for an appropriate level of Hume, and an assessment of the impact of the design decisions in this workpackage on certificate size and validation time. Should the work in WP3, WP4 and WP8 suggest an approach to certificate generation that is superior to our PCC infrastructure in the MRG project, we will aim for a prototype compiler module implementing this new approach.

We have informally described certificate generation for Core-HUME expression. Core-HUME is the intermediate language used by the compiler, and it is the same language on which resource analysis

is performed. Since our approach to certificate generation is closely linked with the high-level, type-based resource inference, this is the appropriate language level. We have discussed certificate size and validation time, in terms of steps in a manually generated certificate, for an example program in Section 3.2. With a fully automated proof strategy, as described in Section 2.2, the size of the certificate could be reduced even further, ideally to just one invocation of the generic strategy. Thus, we can provide very compact certificates, at the cost of using a fairly heavy-weight validation machinery, namely Isabelle/HOL itself. The latter also has an impact on the validation time for a certificate. However, in the presented examples, validation is not excessively expensive and only takes a few seconds. Most notably, no extensive proof search is performed by our proof strategy. The highest potential cost is attached to solving inequalities over resource variables, as discussed in Section 3.3, which might require a search over a narrow set of natural numbers. Our approach to certificate generation mainly follows our earlier work in the MRG project, albeit on a richer language. Therefore, we haven't implemented a compiler module for certificate generation. Section 2.2 discusses a general proof strategy for verifying resource bounds. The list-copy example in Section 3.2 and the control code example in Section 3.3 demonstrate that a proof can be created out of the high-level type-inference. The full proof scripts for both examples are given in Appendix A.1 and A.2. The former also discusses the link between the generated proof script and the high-level type-inference. Most of the effort in this workpackage has been put into formalising the program logic and the resource logic for Core-HUME in Isabelle/HOL. These formalisations represent a powerful reasoning infrastructure, and can be used as an interactive verification workbench for manually developing certificates of bounded resource consumption.

4.2 Positioning of this Deliverable

Figure 3 gives an overview of the dependencies between workpackages, with the current deliverable D31 in WP6c 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], and on the certificate format and the resource logic in Deliverable D21 [LB08].

The scope of this deliverable is significantly narrower than initially envisioned: we used simpler example programs, and only one branch of a control box of a full Hume application. The reason for this limited scope is the reduced effort spent in WP6, in favour of more effort spent on the core workpackages WP3 and 4 on resource analysis. This shift in efforts has been reported on in the reviews for years 2 and 3 and was agreed on by the reviewers. However, despite the reduced effort we fully achieved the goals in tasks WP6a and WP6b, by formalising a full program logic and a full resource logic in Isabelle/HOL. This formalisation exceeds the planned "prototype formalisation in a theorem prover ... of key, novel Hume language features," as specified in the success criterion for WP6b. We believe that these formalisations represent a valuable platform for further research on the formal basis of Hume, in general, and on proving and verifying resource bounds for Hume expressions, in particular.

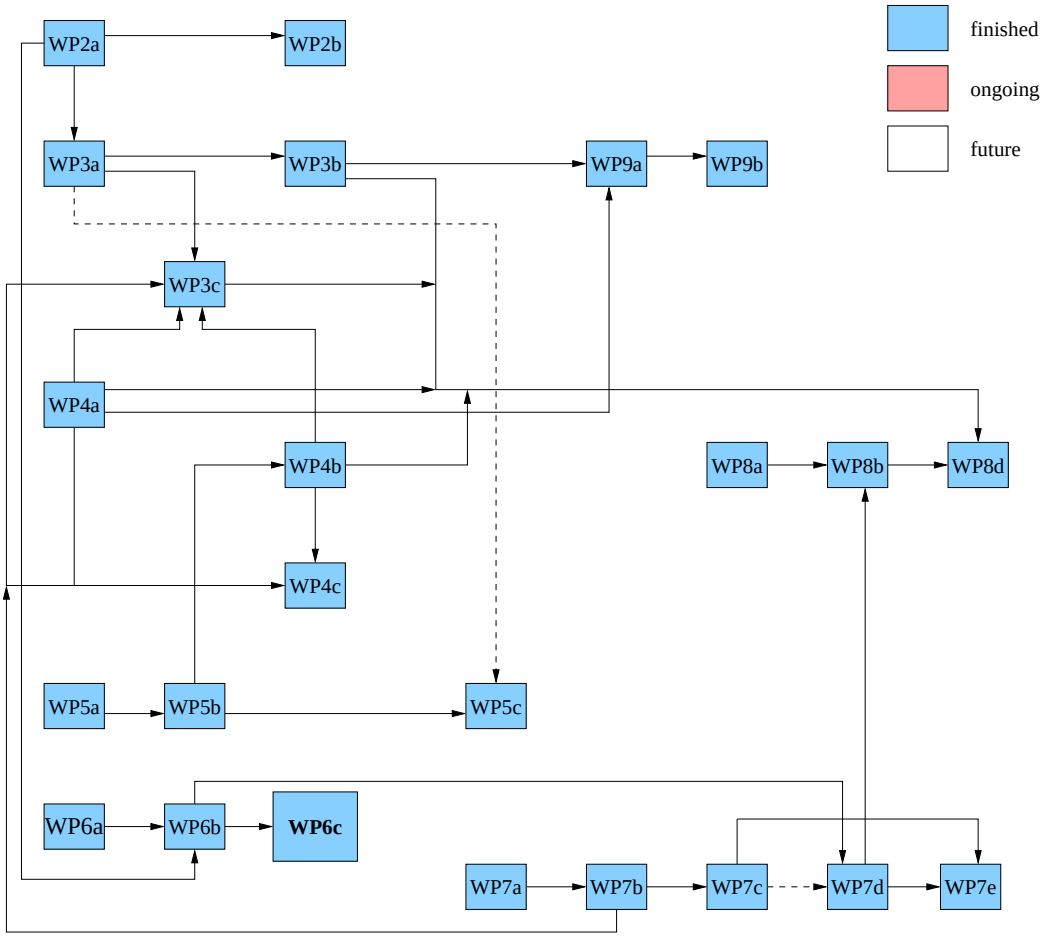

Figure 3: Project Task Dependencies

References

[Hof00] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000. 1

[JLH07] S. Jost, H-W. Loidl, and K. Hammond. Report on Heap-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D11. 1

[LB08] H-W. Loidl and L. Beringer. Certificates. EmBounded Project Deliverable, July 2008. Deliverable D21. 1, 2.1, 4.2

[LG07] H-W. Loidl and G. Grov. Assertion Language. EmBounded Project Deliverable, October 2007. Deliverable D17. 2.1, 4.2

[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. 2.2

A Isabelle/HOL Proofs of Resource Bounds

A.1 Case study: List-copy

This appendix gives the full Isabelle/HOL proof of heap bounds for the list-copy exampled in Section 3.2, and relates it to the information generated by the high-level type inference. The former has already been described in D21, and here we focus on the aspect of certificate generation. The latter, is driven by the order in which type inference is performed. We add a trace of this order at the end of this appendix.

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
```

For the purpose of certificate generation the structure of the above proof and its relationship to the high-level type-inference are relevant. Below we show a trace of the rules used in the high-level type inference when inferring the heap consumption of the list-copy program. It shows, that the order in which the inference rules are applied is the same as the order of the rules in the resource logic. Thus, we can in principle generate a certificate by turning this trace into an Isabelle/HOL proof script, verifying the resource assertion that is read off the high-level type of this function. The only steps missing are the injection of calls to the simplifier at the right points in this proof script.

In more detail, comparing the trace below with the proof of lemma `copy1DA`, the type rule `EXPR - FCASE` and the corresponding `PATT - ...` rules correspond to `DACaseOnePCon8` in the proof script (which is DACase in the rules in D21). The type rule `EXPR - CON` represent a constructor application, corresponding to `DAConstr8` in the proof script. The non-trivial pattern of a cons-cell shows up in the trace as `PATT - PCon` followed by 2 `PATT - PVar`. In the rules of the resource logic, the constructor pattern is directly encoded into the `DACaseOnePCon8` rule. Then the right-hand-side of the constructor

pattern is processed. This involves applying the type rule for let `EXPR - GLET`, corresponding to `DALet` in the proof script. Then a sequence of bindings follows, with a function call and a variable on the right hand sides. The (exact) function call, `EXPR - ExactApp"copy"`, shows up in the proof script as `EmptyProofInvs`, using our proof strategy involving the *goodContext* predicate as discussed in Section 2.2. The second binding only binds the pattern variable `x`, applying `EXPR - VAR`, showing up in the proof script as `DAVar8`. Finally, the body of the let is a constructor application. Thus, the type rule `EXPR - CON` is applied, which corresponds to the rule `DAConstr8` in the resource logic.

```
DBGTRACE_InfRules written Thu Aug 28 19:00:51 CEST 2008

L0021C024: EXPR - FCASE ?copy_arg_11
L0022C016: HPAT - PCon "?N1int" []
L0022C016: PATT - PCon "?N1int" []
L0023C007: EXPR - CON "?N1int"
L0025C022: HPAT - PCon "?C1int" [PVar "x",PVar "xs"]
L0025C022: PATT - PCon "?C1int" [PVar "x",PVar "xs"]
L0025C022: PATT - PVar "x"
L0025C022: PATT - PVar "xs"
L0030C006: EXPR - GLET "?bdg_xs_1" "?z_2" "?bdg_x_3"
L0027C020: EXPR - VAR xs
L0028C015: EXPR - ExactApp"copy""?bdg_xs_1"
L0028C022: ARGS - "?bdg_xs_1"
L0029C019: EXPR - VAR x
L0030C009: EXPR - CON "?C1int"
L0030C009: ARGS - "?bdg_x_3"
L0030C009: ARGS - "?z_2"
```

A.2 Case-study: Control code

This section gives the full Isabelle/HOL proof of heap bounds for the control example. A detailed discussion of the proof structure is presented in Section 3.3.

```
lemma robucar5DA: "{} \<rhd> robucar5Body :
                              resDA (Suc (Suc 0)) 0 {xvar,zconst} robucar5DACtxt
                                    (RefETy (0::nat) (ConETy OutTy [0,0] []))"
apply (simp add: robucar5Body_def)
apply (rule DACaseOnePCon8)
apply (simp add: fmapLookup_aux)
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (rule conjI) apply simp+
apply (simp add: conSigE0_TIN)
apply (simp add: conId_TIN)
apply simp+
apply (simp add: vars_unique_def)+
apply (rule DALet)
apply (rule DAPrimBin)
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply (simp only: equals_2_def)
apply (simp add: basicEType_aux)
apply (rule DAIf)
apply simp
apply (rule DALet)
```

```
apply (rule DAValue)
apply (simp add: etypeof_val_def basicEType_def)
apply (simp add: etypeof_val_def basicEType_def)
apply (rule DALet)
(* CallFunExp updateFun ... *)
apply (subgoal_tac "{} \<rhd> CallFunExp (FN ''updateFun'') [VN ''vvar'', VN ''k2var'', VN ''wvar''] :
                            the_funSpecTab (FN ''updateFun'') [VN ''vvar'', VN ''k2var'', VN ''wvar'']"
  prefer 2
  apply (rule CallFunByGoodCtxt)
  apply (insert robucar5good) apply simp
  apply (simp add: robucar5Ctxt_def)
  apply (simp add: robucar5Ctxt_def)
  apply (rule conjI) apply simp
  apply simp
  apply (simp add: the_funSpecTab_updateFun)
(* done with CallFunExp *)
apply (rule DAWeak_Bndk)
apply fastsimp?
apply simp+
apply (rule DALet)
apply (rule DALet)
apply (rule DAPrimBin)
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply (simp only: equals_9_def)
apply (simp add: basicEType_aux)
apply (rule DAIf)
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
(* CallFunExp wvtow16_le ... *)
apply (subgoal_tac "{} \<rhd> CallFunExp (FN ''wvtow16_le'') [VN ''x1var''] :
                            the_funSpecTab (FN ''wvtow16_le'') [VN ''x1var'']")
  prefer 2
  apply (rule CallFunByGoodCtxt)
  apply (insert robucar5good) apply simp
  apply (simp add: robucar5Ctxt_def)
  apply (simp add: robucar5Ctxt_def)
  apply (rule conjI) apply simp
  apply simp
  apply (simp add: the_funSpecTab_wvtow16_le)
(* done with CallFunExp *)
apply (rule DAWeak_Bndk)
apply fastsimp?
apply simp+
(* CallFunExp wvtow16_be ... *)
apply (subgoal_tac "{} \<rhd> CallFunExp (FN ''wvtow16_be'') [VN ''x1var''] :
                            the_funSpecTab (FN ''wvtow16_be'') [VN ''x1var'']")
  prefer 2
  apply (rule CallFunByGoodCtxt)
  apply (insert robucar5good) apply simp
  apply (simp add: robucar5Ctxt_def)
  apply (simp add: robucar5Ctxt_def)
  apply (rule conjI) apply simp
  apply simp
  apply (simp add: the_funSpecTab_wvtow16_be)
(* done with CallFunExp *)
```

```
apply (rule DAWeak_Bndk)
apply fastsimp?
apply simp+
apply (rule fmapDom_aux)
apply simp+
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply simp+
apply (rule DALet)
apply (rule DAValue)
apply clarsimp?
apply (simp add: etypeof_val_def basicEType_def)
apply (simp add: etypeof_val_def basicEType_def)
apply (rule DAConstr8)
apply (rule argTypes_aux)
apply (rule conSigE0_TOUT)
apply simp+
apply (rule conjI) apply simp
apply (rule conjI) apply simp
apply simp+
apply (simp add: conId_TOUT)
apply simp+
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply (simp add: vars_unique_def)
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply simp+
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply simp+
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply simp+
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply simp+
apply (rule DAWeak_Bnd)
apply (rule DAConstr8)
apply clarsimp
apply (rule conSigE0_TOUTNOT)
apply simp+
apply (rule conjI) apply simp
apply (rule conjI) apply simp
apply simp+
apply (simp add: vars_unique_def)
apply simp+
apply (simp add: conId_TOUTNOT)
apply (subgoal_tac "Suc 0 \<le> Suc 0")
apply assumption
apply simp+
apply (simp add: dom_fmap_upds robucar5DACtxt_def)
apply simp+
apply (rule DAWeak_Bnd)
apply (rule DAConstr8)
apply clarsimp
apply (rule conSigE0_TOUTNOT)
apply simp+
apply (rule conjI) apply simp
apply (rule conjI) apply simp
apply simp
```

```
apply (simp add: conId_TOUTNOT)
apply simp+
apply (simp add: vars_unique_def)
apply (subgoal_tac "Suc (Suc 0) \<le> Suc (Suc 0)")
apply assumption
apply simp+
apply (rule usageSet_aux)
apply simp+
done
```