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

Specific Targeted Research Project (STReP)
FET Open

D13 (WP4): Prototype Implementation of space analyses (Revised)

Due date of deliverable:	30th April 2006
Actual submission date:	20th February 2007

Start date of project: 1st March 2005 Duration: 36 months

Lead contractor: St Andrews University Revision: 1.25

Purpose: To describe the implementation of the Stack- and Heap-space analyses that have been formally defined in Deliverables D5 and D11.

Results: We have implemented a fully automatic HUME Stack- and Heap-space analysis, which allowes us to efficiently determine the space requirements of HUME programs in relation to their input size.

Conclusion: Successfully accomplishing this deliverable has enabled us to test our space analyses against a number of sample HUME programs. These have, in turn, deepened our understanding of these nature and quality of the analyses, leading us to refine them further and explore new features. Some of these insights have already been incorporated in the design of the revised formal analyses described in Deliverables D5 and D11.

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)	

Prototype Implementation of space analyses (Revised)

Author: Steffen Jost <jost@cs.st-andrews.ac.uk>

Abstract

This document describes how to use the prototype implementation of the Heap- and Stack-space analyses described in Deliverables D5 and D11 (both WP4). We discuss where the prototype implementation and our theoretical work may differ, both in technical essentials and through the implementation of experimental features. Since execution of the analysis prototype on various program examples for determining Stack- and Heap-space usage is already detailed in the deliverables describing the underlying mechanics (D5 and D11, respectively), in this document we will focus entirely on those details that are concerned with the implementation.

Contents

1 Overview and Basic Usage

We have implemented the Heap- and Stack-space analysis as detailed in Deliverables D5 and D11. We have created a stand-alone analysis tool, which processes programs written in Core-HUME (see section 2), writes the constructed set of linear constraints in a human-readable format to a file, calls a standard linear programming solver (*lp-solve* [1]) to solve the constraints and returns the interpreted solution in text form. Our implementation was written in Haskell, a modern, powerful functional language. This both has productivity benefits (up to an order of magnitude compared with C/Java), reduces code maintenance costs, and allows straightforward integration with other HUME tools.

The analysis implementation currently comprises 22 modules containing 7387 lines of code. Compilation requires the following portable and freely-available tools: *The Glasgow Haskell Compiler* [3] (version 6.4.2), the Haskell parser generator *Happy* [5] (version 1.5); and the the lexical analyser generator *Alex* [2] (version 2.0.1). Running the analysis also requires the linear programming solver *lp_solve* [1] (version 5.5), which is also freely available. Pre-packaged linux binaries are provided on the EmBounded Project Website, `http://www.embounded.org` [7], and we have provided an open-access web interface to the analysis at `http://www.embounded.org/software/cost/cost.cgi`. Section 6 describes the usage of the standalone binary analysis.

We have found our implementation to be highly efficient in practice. In fact, the runtime required to analyse most of the small program examples discussed in Deliverables D5 and D11 was too small to measure with standard tools on a 1.7GHz Pentium laptop, with 1GB main memory and 2MB cache. For the largest example that we have studied to date, the drinks vending machine (Section 4.4 of Deliverable D5; Section 5.4 of Deliverable D11), the translation from HUME code to Core-HUME requires 0.03 seconds on the same machine. Generating and solving the constraint set for *either* Stack- or Heap-space usage requires about 0.36 seconds, of which 0.05 seconds were used by the linear programming solver. While already acceptable, we anticipate that the total runtime could be reduced by eliminating the intermediate files that are currently used to convey information between the compiler and the analysis. Based on experience of an earlier program analysis using linear programming [4], we expect the runtime to scale well to larger program examples. For the program examples discussed in Deliverables D5 and D11, the number of constraints generated by the Heap-space analysis was as follows.

	Lines of Code		**Heap-space analysis**	
Program Example	**HUME**	**Core-HUME**	**Constraints**	**Variables**
Factorial	10	23	16	21
List Summation	14	21	10	14
Iterated Multiplication	37	64	118	130
Drinks Vending Machine	64	138	480	373
Canny Edge Core Code	53	78	100	127

similarly, the Stack-space analysis generates the following numbers of constraints:

	Lines of Code		**Stack-space analysis**	
Program Example	**HUME**	**Core-HUME**	**Constraints**	**Variables**
Factorial	10	23	18	23
List Summation	14	21	12	16
Iterated Multiplication	37	64	142	154
Drinks Vending Machine	64	138	546	435
Canny Edge Core Code	53	78	134	151

2 Translation to Core-HUME

The source to the analysis is required to be in the Core-HUME format specified in Deliverable D11. Core-HUME supports all essential language features such as full recursion and higher-order functions, but requires the removal of some syntactic sugar, e.g. by transforming the program into *let-normal form* (defined in Deliverable D11). It follows that Core-HUME is orthogonal to the HUME-Layers, which provide a custom balance between expressivity and cost-control. Programs in Core-HUME are not intended to be compiled or executed: the sole purpose of Core-HUME is to allow us to determine the cost of executing the *original* HUME program.

In order to support our analysis, the implementation of the *Prototype Hume Abstract Machine Compiler* (phamc) [6] has been extended to automatically translate ordinary HUME programs into Core-HUME using the `-r` flag. In contrast to the description given in Deliverable D11, the Bundle- and Wire-wrapper types that are introduced to the translation are not unique, but carry automatically generated names derived from the types they contain. For example, a bundle type containing two wires of type `int` and one of type `char` will receive the name `B3intintchar`. This is sufficient to allow the standard typing defined in Deliverable D11. Note that if the analysis is run to determine Heap-space usage (only), then it is currently necessary to manually rename wires, in order to simulate the resetting of the box heap.

A further simplification is performed by transforming all source-level over-applications using a series of standard function applications connected by a ghost-let. The precise cost is preserved by attaching a flag to each application expression, which is used to indicate whether the application was originally part of an over-application. expression in the source HUME program or not. In this way, we avoid introducing fresh expression-level variables, as described in the Over App and Over App Var rules of Deliverables D5/D11 (Sections 3 and 4, respectively)..

3 Implementation of formal Type Rules

We will now describe the implementation of the formal type rules given in Deliverables D5 and D11. As usual with Haskell, the code follows the formal definition fairly closely, though a few important changes had to be made. The most important of these is the expansion of our compressed formal notation to give a precise meaning in the Haskell program. For example, the rule Conditional for Stack-space usage

$$\frac{\Gamma \vdash^{p}_{p'} e_t : A \mid \Phi \qquad \Gamma \vdash^{p}_{p'} e_f : A \mid \Psi}{\Gamma, x{:}\mathtt{bool} \vdash^{p-1}_{p'} \mathtt{if}\ x\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f : A \mid \Phi \cup \Psi} \qquad (\textsc{Conditional})$$

was defined as short-hand for:

$$\frac{\Gamma \vdash^{q_1}_{q'_1} e_t : A \mid \Phi \qquad \Gamma \vdash^{q_2}_{q'_2} e_f : A \mid \Psi}{\Gamma, x{:}\mathtt{bool} \vdash^{q_0}_{q'_0} \mathtt{if}\ x\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f : A \mid \Phi \cup \Psi \cup \left\{\begin{matrix} q_0 = q_1 - 1, & q'_1 = q'_0, \\ q_0 = q_2 - 1, & q'_2 = q'_0 \end{matrix}\right\}} \qquad (\textsc{Conditional}^*)$$

Since the linear combinations of resource variables and constants above and below each turnstile ($\vdash$) must match when constructing the proof tree of a type derivation, the implementation must introduce fresh intermediate variables and add constraints to link these new intermediate variables with those that are given.

We also took care to make the substructural rules explicit in our theoretical description of the analyses, since this simplified the process of reasoning about the rules by separating recurring aspects into separate rules. However, in the implementation, it is difficult to decide when to apply the substructural rules, since by their nature they are not syntax-driven. We therefore apply substructural rules at all points where they *might* be used, since it is not semantically harmful to apply them more often than required (though it may have an impact on performance). It is then convenient to merge two successive rules together into a single step within the implementation. For example, considering rule Var for Stack-space usage:

$$\frac{}{x{:}A \vdash^{1}_{0} x : A \mid \emptyset} \qquad \text{(Var)}$$

followed by an application of Relax Stack

$$\frac{\Gamma \vdash^{q}_{q'} e : A \mid \psi}{\Gamma \vdash^{p}_{p'} e : A \mid \psi \cup \{p \geq q, p - q \geq p' - q'\}} \qquad \text{(Relax Stack)}$$

to yield the proof tree

$$\frac{\frac{}{x{:}A \vdash^{1}_{0} x : A \mid \emptyset}}{x{:}A \vdash^{p}_{p'} e : A \mid \{p \geq 1, p - 1 \geq p' - 0\}}$$

These two rules can be directly combined into the single rule

$$\frac{}{x{:}A \vdash^{p}_{p'} x : A \mid p \geq p' + 1} \qquad \text{(Var+Relax)}$$

since all variables range only over the non-negative rational numbers. This combined rule can then be implemented by:

```
-- VARIABLE
iexpression_in_ctxt ctxt din anno expr@(VarExp v) dout =
  do
    cns <- domina "Var" din [dout] [RCpvar]
    return (ctxt!v,cns)
```

where `din` is the variable above the turnstile (p in the rule), `dout` is the variable below the turnstile (p' in the rule), and the function `domina` generates the required linear constraints saying that the variable given as the second argument is greater than the sum of all variables in the third argument plus the sum of constants in the fourth argument (with the first argument just being a label). Since the third and fourth arguments are singleton lists, we have `din` $\geq$ `dout` + `RCpvar` as required, since the constant `RCpvar` is defined as 1 if the analysis was invoked for Stack-space usage. Similarly, all equalities within the previous example of rule Conditional* become inequalities ($\geq$).

4 Reading Constraints

Applying the implementation to a program that conforms to the Core-HUME format of Section 2 will do one of three things:

a) If an ordinary HUME expression is given *after* all function and box declarations, then the annotated type of this expression *alone* is inferred. This represents a simple yet useful way to determine the resource consumption of certain parts of a program.

b) If at least one box declaration was given, then the annotated types for all declared boxes are inferred.

c) Otherwise, the annotated types of all globally defined functions are inferred.

In addition, all constraints arising from the complete program analysis are saved in a file that readable by the `lp-solve` application (usually `constraints.lp` under Unix), even if the linear programming solver is unable to find a solution to those constraints. The file contains a comment containing some statistics, the objective function and a list of linear constraints. Each constraint starts with a label, containing the line and column number of the expression within the Core-HUME program which triggered the generation of this particular constraint, as well as a four-letter code giving more information about the constraint. For example,

```
/*
  L0026C27:
    ilist[d14|d16;int,#]
  subtypes
    ilist[d08|d09;int,#]
*/
L0026C27_Sub: + d09 - d16  <= 0;
L0026C27_Sub: + d08 - d14  <= 0;
/*
  L0026C27: Application 'insert'
            (int,ilist[d08|d09;int,#]) -x07/y07-> ilist[b07|b08;int,#]
*/
L0026C27_Api: + x07 - z03 + e01  <= 0;
L0026C27_Apo: - y07 + z04 - e01  <= 0;
```

indicates that either rule Subtype or rule Supertype was applied to the given type in line 26, with the resulting constraints being given in the next line. The implementation does not distinguish between Subtype and Supertype since we use the same mechanism to restrict one type to another in both cases. We can also see that the function `insert` is applied in line 26 and column 27 of the input program. This is the cause of the subtyping, since the context held a different type for the argument (`ilist[d14|d16;int,#]`) than was required as an argument to the function (`ilist[d08|d09;int,#]`). The function application itself generates the inequalities $z_3 \geq x_7 + e_1$ and $y_7 \geq z_4 - e_1$ arising from the refined type rules of Section 3.

$$\frac{\Sigma(\mathit{fid}) = \big(\_;\_;\forall\alpha \in \psi.A_1,\ldots,A_a \xrightarrow[n']{n} C\big) \qquad k \geq 0 \qquad k = a}{y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{m}_{m'} \mathit{fid}\ y_1 \cdots y_k : C \ \big|\ \big\{ m \geq n, \quad m - n \geq m' - n' \ \big\}} \ (\text{App* (Heap)})$$

which can be specified equivalently as:

$$\frac{\Sigma(\mathit{fid}) = \big(\_;\_;\forall\alpha \in \psi.A_1,\ldots,A_a \xrightarrow[n']{n} C\big) \qquad k \geq 0 \qquad k = a}{y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{m}_{m'} \mathit{fid}\ y_1 \cdots y_k : C \ \big|\ \big\{ m \geq n + q, \quad m' \leq n' + q \ \big\}} \ (\text{App** (Heap)})$$

by introducing a fresh resource variable $q \in \mathbb{Q}^+$ (`e01` above). It is thus relatively easy to track each constraint back to both the originating program expression and to the corresponding type rule.

Whenever the constraint set is solvable our implementation will write the file `constraints.solved` to the current directory. This file is identical to `constraints.lp`, except that all variables have been replaced by the concrete values that have been inferred by the linear programming solver. Any constraints that are strict inequalities will be marked, since they correspond to a computation path where resources have been discarded using the Relax rule. For example, the two possible branches of an if-expression may have a different resource usage, so the entire if-expression is awarded the worst-case cost. At least one of the constraints generated for the branch that has the lower resource consumption will then have a strict inequality. The line tag then allows the programmer to quickly determine alternate computational branches of a program which have differing resource costs.

5 Experimental Features

The prototype implementation includes two experimental features, which we anticipate will further improve the resource bounds we obtain.

Potential Carrying Floats The analysis described in Deliverable D5 and D11 only allows potential within non-base types. We are currently investigating whether it is feasible and useful to allow potential to also be associated with base types such as floats or integers. Since our testbed HUME progam examples (Deliverable D7, WP8) frequently use floats, we decided to test the idea with floats. The `float` type is consequently annotated with a resource variable, written `float<q>`, whose value multiplied with the value of the float represents the potential carried by a float of this type, analogously to the potential of a list type. Please recall that although the notion of potential refers to runtime values, no calculations are done at runtime: all potential calculations occur entirely within the compiler. The potential of a float is bound and released whenever a constant value is added or subtracted from the float of the annotated type, or when the float is matched to a constant value. Other operations, such as multiplication and division, cause the float to lose its annotation. The analysis distinguishes betwen those floats which might carry potential and those which may not: floats of type `float<0>` and `float` carry no potential, but the latter was not even allowed to carry any potential. Since a negative potential cannot occur, we have also included a very basic and currently incomplete value-range analysis. It follows that the derived resource bounds can only be trusted if all floats which carry potential are known to be non-negative at runtime. For example, the factorial function shows one safe use:

```
fac x =
  if x <= 1.0
  then 1
  else x *. (fac (x -. 1.0));
```

since potential is only released in the else-branch, where we know the value of `x` to be non-negative, even after subtraction. This program example is further discussed in Deliverable D11. Note that, since this experimental feature is generally unsafe to use, it can be disabled by using the `-nfp` option to the implementation.

Potential Capturing Closures Since function closures may be used repeatedly, we must take care to avoid any unlawful duplication of potential. Hence the annotated type rules for under-applications in Deliverables D5 and D11 require that all captured types are potential-free by asserting that they can be shared indefinitely. While this restriction is certainly safe, it might also be overly restrictive. We are therefore investigating the possibility of using a standard flow-analysis to determine the maximum number of calls to a function closure. Where such a bound exists, the arguments used in the under-application that produce the closure can be pre-shared to the specified bound, allowing them to carry some potential. While we anticipate that use of this feature might actually result in less precise bounds, it might also allow a wider range of HUME programs to be costed. The feature is fully implemented, but since the flow-analysis for determining the maximum use of a closure is not yet finished, manual intervention is required for it to come into effect.

6 Usage

The analysis is invoked under Unix by `art3 [options] [infile]`. If no input file is specified, then the program to be analysed is read from the standard inpiut. The implementation currently recognises only the fairly limited set of command-line options given below.

`-S`, `-RKStack` Performs Stack-space usage analysis.

`-H`, `-RKHeap` Performs Heap-space usage analysis [the default analysis].

`-T`, `-RKTime` Performs Worst-Case Execution Time analysis using the *default* time metric (equivalent to `-RKTimeM32`).

`-RKTimeM32` Performs Worst-Case Execution Time analysis in terms of clock cycles required for the Renesas M32C/85 processor.

`-nfp` Disables the experimental feature of assigning a potential to floating-point types (Section 5).

`-v`, `-V` Verbose output for the preprocessing stage (only), where `-V` is more verbose than `-v`. Note that information from the analysis is currently written to the file `DBGTRACE`.

`--version` Displays version information.

`-h`, `--help` Prints usage and version information, plus the list of known resource kinds.

If multiple resource types are specified, then only the last will actually be analysed.

References

[1] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve: Open source (mixed-integer) linear programming system. GNU LGPL (Lesser General Public Licence). `http://lpsolve.sourceforge.net/5.5`.

[2] Chris Dornan, Isaac Jones, and Simon Marlow. Alex: A lexical analyser generator for haskell. `http://www.haskell.org/alex`.

[3] The Glasgow Haskell Compiler. `http://haskell.org/ghc`.

[4] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.

[5] Simon Marlow and Andy Gill. Happy - the haskell parser generator. `http://www.haskell.org/happy`.

[6] Prototype hume abstract machine compiler.

[7] Robert Rothenberg. Website of the embounded project (ist-510255). `http://www.embounded.org`.