AHA
Amortized **H**eap Space Usage **A**nalysis

Marko van Eekelen, Bart Jacobs, Erik Poll, and Sjaak Smetsers
Radboud University Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
Contact: marko@cs.ru.nl

August 2005

1c. Principal Investigator

Dr. Marko van Eekelen. (http://www.cs.ru.nl/~marko)

2a. Summary

This project involves research into an amortized analysis of heap-space usage by functional and imperative programs. Estimating heap consumption is an active research area since it becomes more and more an issue in many applications. Examples include programming for small devices, e.g. smart cards, mobile phones, embedded systems and distributed computing, e.g GRID computing. The standard technique for estimating heap consumption gives in many cases unrealistically high bounds. Therefore, in practice amounts of heap are used that are unnecessarily expensive and for small devices highly unpractical. A more accurate analysis is wanted for these cases in particular, and for high integrity real-time applications in general.

Amortized analysis is a technique which is used to obtain accurate bounds of resource consumption and gain. For the amortization analysis of a resource one considers not the worst case of a single operation but the worst case of a sequence of operations. The overall amortized cost of a sequence is calculated by taking into account both the higher costs of one operation and the lower costs of another weighing them according to their distribution. In many cases amortized analysis can give rise to much more accurate resource consumption estimates than the standard worst case analysis.

A combination of amortization and type theory allows to check linear heap consumption bounds for functional programs with explicit memory deallocation. Recently, substantial progress has been made showing that the method can be adapted to deal with *non-linear* bounds for programs over lists written in a strict functional programming language.

This project proposes to extend this method to a lazy functional language as well as to transfer the results of the functional programming community to the imperative object-oriented programming world by applying the amortized method to derive accurate bounds for heap usage of Java programs. In this way the potential impact of amortized analysis is increased significantly.

2b. Abstract for laymen (in Dutch)

Dit project onderzoekt de mogelijkheden om te analyseren hoeveel geheugen een programma gebruikt, met als uiteindelijke doel om, vóór een programma geëxecuteerd gaat worden, te kunnen voorspellen hoeveel geheugen een programma gaat gebruiken. Dit is vooral van belang voor computers met zeer beperkte geheugencapaciteit, zoals chipkaarten of mobiele telefoons, waar problemen op kunnen treden als tijdens executie van een programma blijkt dat het beschikbare geheugen niet toereikend is. Daarnaast kan het gebruikt worden om de effecten van optimalisaties geïntroduceerd bij de compilatie van programma's beter te analyseren.

3. Classification

Computer science.
Subdisciplines: 6. Fundamenten: 6.1 Complexiteitstheorie, 6.5 Formele methoden
1. Computer- en netwerksystemen: 1.3 Dependability
3. Software engineering: 3.2 Specificatiemethoden

The NOAG-i themes most relevant to this research is 'Methoden voor Ontwerpen en Bouwen'. Enhancing specification and analysis techniques a contribution is made to attacking the challenge of 'Quality by Design'.

4. Composition of the research team

Name	Specialism	Involvement (fte)
dr. Marko van Eekelen	functional programming/semantics, program analysis	0.1
dr.ir Erik Poll	program logics, type theory, JML, Java	0.1
dr. Sjaak Smetsers	functional programming, type systems for functional languages, compiler technology	0.1
Prof.dr. Bart Jacobs	semantics, type theory, Java, use of proof tools (esp. PVS)	p.m.
project postdoc (candidate dr. Olha Shkaravska)	complexity theory, automated theorem proving	1.0
project PhD student (no candidate)		1.0

5. Research School

The research group participates in the national research school *Institute for Programming research and Algorithmics* (IPA). The proposed research falls within IPA's main theme *Formal Methods*, more specifically in the sub-themes *Semantics* and *Specification, verification and testing*, and the theme *Software Technology*, more specifically in the sub-themes *Construction process and architecture*.

The research also addresses one of the four application areas selected by IPA, namely *Security*, more specifically the topics *Software security* and *Smart cards*: memory usage of software is an important security concern for devices with limited memory such as smart cards, as a simple way for malicious (or incorrect) code to succeed in a denial-of-service attack is to exhaust all available memory.

6. Description of the proposed research

Since memory exhaustion will invoke garbage collection, manipulations over a heap indirectly slow down execution and hence influence time complexity. A heap usage analysis for a language with explicit heap-cell deallocation will therefore enable a more accurate estimation of time consumption. With an amortized analysis we aim for a fairly accurate estimate.

In an amortized analysis [18, 20] one considers *credits* and *potentials*. A *credit*, which is the (nonnegative) difference between an amortized and the actual cost of an operation, is used to cover the resource consumption by other operations. A *potential* is the sum of the credits associated with an entire data structure.

In [11] the authors present an annotated type system which allows an amortized analysis inferring linear bounds on a heap-space consumption/gain for strict first-order functional programs with explicit memory deallocation. The types are annotated with rational constants playing a role of coefficients for the linear bounds, considered as functions of the size of input/output.

In 2005 [21] it was shown that the method may be adapted for non-linear bounds for first-order functional programs over polymorphic lists.

The type-checking/inference is reduced to a consistency-check/solving of a set of constraints which are numerical side conditions collected from the type-derivation tree for a program. If the system of the constraints is consistent then the program is type correct. Type inference can be achieved if the system of the *parametrical* constraints is solvable.

Annotations in the type system are credits for constructors. The typing rules are designed in such a way, that annotations are always nonnegative functions and overall amortized cost/gain is not less/more than an actual cost/gain. If annotations are incorrect, then the type-checking procedure fails due to inconsistency of the corresponding set of constraints – see [21] for an example.

Amortized Analysis

For the amortization of a resource one considers not single operations but their combinations. This defines a mechanism of obtaining accurate bounds for the consumption.

As a very small example consider the composition of two functions, f and g, such that $range(g) \subset dom(f)$ where f and g are defined as follows:

```
f x  =  match x with  Nil ⇒ cons(1, Nil)
                    || cons(h, t) ⇒ cons(1, cons(2, Nil))

g x  =  Nil
```

Trivially, in the worst-case f consumes 2 heap units.

The type system from [21] is able to infer that the heap consumption for f, is defined by the following bound function T:

$$T(l) = \begin{cases} 1, & l = 0 \\ 2, & l \geq 1 \end{cases}$$

where l is the length of an input list.

Using the bound T an amortized analysis easily shows that $f(g(x))$ always consumes 1 heap unit, improving on the worst case analysis.

Another case where amortized analysis is profitable is for calculating the heap consumption of a vector array: instead of multiplying the worst case for each element a careful analysis can take all ups and downs of different kinds of elements into account and achieve a lower overall estimate.

Proposed Research Lines

We propose to investigate on one hand the extension to lazy evaluation of the method of [21] and on the other hand the transfer of this method to an imperative setting. In this way the project both enhances fundamental theory and practical impact.

6a. Research questions and expected results

The proposed project investigates the possibilities for analysing heap usage both for functional and imperative object-oriented languages, more specifically Clean and Java. It aims to answer the following research questions:

- It is clear, that the heap analysis for functional languages can be improved so that a wider class of resource usage bounds than just linear bounds can be guaranteed. The question is how complex the type-checking and inference procedures may be. In particularly, which arithmetics and constraint solvers will be needed and for which classes of programs?

- Can heap space analysis be done for lazy functional languages?

  Heap space analysis for lazy functional languages is clearly more complicated than for strict languages, because the heap space is also used for unevaluated expressions (closures). The amount of memory that is used at a certain moment depends on the evaluation order of expressions, which in its turn is influenced by the strictness analyser of the compiler that generates the code.

- How successfully can one adopt the approach for object-oriented imperative languages? The ultimate aim here is to be able to prove – or, better still, derive – properties about the heap space consumption for Java programs specified in an extension of JML (Java Modeling Language) [15, 8].

The expected results are:

- Accurate and expressive analyses for strict and lazy functional languages, that can guarantee a wider range of heap space usage properties. The functional language Clean will serve as a test-bed.

- Techniques to verify heap space usage constraints for object-oriented imperative programs, in particular of Java programs, incorporated into a tool for program verification. Extensions of the program specification language JML to express such constraints.

6b. Methodology

The project will have two research lines.

- A fundamental theoretical one in which amortized heap analysis with non linear bounds is extended to a lazy language.
- A practical line in which the theoretical results are transferred to a more practical imperative object-oriented setting.

Below, for both research lines we describe and motivate the methodology, the starting point analysis and the language which is chosen as a vehicle for expressing programs.

6b1. Towards Amortized Heap Analysis of a Lazy Language

An amortized time analysis for call-by-need [1] languages is considered in [18]. Instead of credits it uses *debts* to cover costs of *closures* (suspensions). A closure is allowed to be forced only after its debt is "payed off" by the operations preceding the operation which forces the closure.

S. Jost has been adopting the results from [11] for higher-order functions, see [13]. The heap-aware inference system takes into account sizes of closures and enable deferred evaluation.

Very recently, O. Shkaravska [21] has adapted this method to achieve non-linear bounds for first-order functional programs over polymorphic lists.

Choice of Programming Language To consider heap usage analysis for lazy functional programming languages, we will begin with a strict version of core-Clean. We have chosen Clean since Clean's uniqueness typing [3] makes Clean more suited as a starting point than e.g. Haskell, since with uniqueness typing reuse of nodes can be analysed in a sophisticated manner.

For this strict Core-Clean language we will define an operational semantics (based on [23]) which will take heap usage into account, and then formulate a (dependent) type system in which annotations in types express costs.

In the context of the European Mobile Resource Guarantees project (MRG) [1], Shkaravska [21] used Camelot [17], an ML-like strict first order functional language with polymorphism and algebraic data types, as a test-bed. To enable analysis of heap usage Camelot makes a syntactic distinction between destructive and non-destructive pattern matchings, where destructive pattern matching allows a node of heap space to be reclaimed; it is expected to be relatively easy to transfer such a distinction to a language that has uniqueness typing, as this can enforce the safe use of destructive pattern matching. Therefore, we expect that the results achieved for Camelot will be quickly transferred to the strict version of core-Clean.

Starting Point Analysis To improve heap space analysis for functional languages beyond guaranteeing linear bounds for memory usage, we generalise the method of [11]. Initially, this will be expressed in a dependent type system, in which each typing rule has numerical side conditions.

[1] Following [18] we associate *call-by-value* with strict languages, *call-by-name* with lazy languages without memoization, and *call-by-need* with lazy languages with memoization.

For instance, in [21], $\mathtt{L}_l(Int,\ k)$ denotes a list of integers of length l such that for its ith element there are $k(i)$ free heap units, i.e. a credit. Here k is a function from natural numbers (without 0) to nonnegative reals.

If an overall heap consumption T is a smooth function on $[\alpha,\ \infty]$, with a bounded derivative T', and $0 \leq \alpha < 1$, one may connect the overall consumption and credits in the following way. To check if a given program consumes $T(x)$ heap units, one needs to perform the type-checking with $k(x) = T'(x)$. We use an approximation of integrals by sums: the total consumption is $\sum_{i=1}^{l} k(i) \approx \int_{i=\alpha}^{l} k(x)\ d\,x = T(x) - T(\alpha)$.

Research Methodology As a next step we will make incremental changes, by changing the strict semantics into a mixed lazy/strict semantics [23] and then investigate the effect on the operational semantics and the type system.

This is not a big step in the dark since the heap-aware inference systems from [11] and [21] already have some flavor of the call-by-need semantics. First, the weakening rule, applied in a backward proof search, allows to exclude from the analysis variables which are not free for an expression under consideration. Second, *shared* usage of variables by several expressions is treated, for instance, in the LET-rule from [21] and the SHARE-rule in [11].

Ultimately, we want to implement the type systems for heap space usage, to obtain an implementation that can check whether a given program, possibly with some type-annotations, meets a given bound on heap space usage, or an implementation that can actually compute such a bound.

6b2. Adaptation to Object-Orientation

Choice of Programming Language As the object-oriented programming languages to be studied we have chosen Java. We will use the Java semantics developed in the LOOP project [12], which includes an explicit formalisation of the heap.

This will first require accurately accounting of heap usage in the type-theoretic memory model underlying the LOOP tool [4].

The Java Modeling Language JML, a specification language tailored to Java, already provides a syntax for specifying heap usage [16], but this part of JML is as yet without any clear semantics. We want to provide a rigorous semantics for these properties about heap space usage and then develop an associated programming logic for proving such properties.

Starting Point Analysis An obvious starting point is the analysis of [21]. The general principle to adjust this for imperative object-oriented (OO) programs is to apply it for classes that admit a functional *algebraic data-type (ADT) interface*, i.e. these classes can be considered as defining a number of operations on an algebraic data type. One extracts basic imperative routines which contain explicit allocation/deallocation, and correspond to (co)algebraic operations, and have functional counterparts, like data constructors and pattern matching(s). A field assignment, for example, may be presented as a composition of the destructive match and a constructor.

Heap-aware typing judgments must be defined for these macro-operations and the language constructs like `if`-branching, sequencing and `while`-repetition. The soundness of the typing judgments is to be proven w.r.t. the imperative operational semantics.

This approach is considered in [5]. Building on the results of [11], the authors investigate the possibility of proving linear bounds of heap usage for an imperative

language Grail, an abstract representation of a Java-like byte code. Imperative programs under consideration are the programs obtained as a result of compilation from a high-level functional language Camelot. These low-level imperative programs are built from the constructs described above, see [5] for more detail.

The idea to adjust a heap-aware annotated type system for functional ADT is sketched in [10]. A time-aware amortized analysis for functional ADT with linear usage of data is investigated in [20]. A simple example of a (co)ADT-like class together with typing rules and a valid specification of a field-updating routine one can find in [21].

Research Methodology With the transferred method of [21] we will consider programming with classes which admit a (co)algebraic data type interface. For instance, programs over binary trees may be composed from null and non-null constructors and case-analysis for the destructor and look-up operations.

These basic routines and imperative constructs have natural counterparts in the functional language. For instance, the case analysis for the destructor corresponds to the destructive pattern matching in the functional language:

```
if (x ==null)
e1;
else { Int a = x.getRoot();
       Tree l = x.getLeft();
       Tree r = x.getRight();
       delete(x);
       e2}
```

The typing rules for constructors and the case-analysis for the destructor and look-up mirror the corresponding ones for the functional language. Their soundness is to be proven using imperative operational semantics.

One can model a field update by the appropriate static swap-method, like for instance,

```
public static PairTree swapLeft(PairTree x)
  {
   if (x ==null)
   return (null);
   else {
        t = x.first;
        newLeft = x.second;
        delete(x);
         if (t ==null)
         return (new PairTree(newLeft, null));
         else { Int a = t.getRoot();
                Tree l = t.getLeft();
                Tree r = t.getRight();
                delete(t);
                Tree y = new Tree(a, newLeft, r);
                return (new PairTree(y, l)) }
        }
  }
```

In JML-annotated programs annotations can be expressed as auxiliary variables, so-called *model fields* [7]. We want to investigate if using suitably chosen model fields,

we can arrive at a programming logic for the verification of resource properties of JML-annotated Java programs.

One of the main problems for heap space analysis will be aliasing. We believe that the results of [2], [14] and Reynold's separation logic [19] can contribute to a solution. Aliasing-aware type systems and logics presented in these works may be considered separately from the resource-aware typing system and are to be combined with it at the very last stage of the design of the proof system. This should improve management of the development. The technique of building such combinations of logics is given in [22].

As a second step, research will be done to alleviate the restrictions that are set upon the classes in order to make the analysis applicable. In order to relax these restrictions, we will investigate the possibility of introducing amortized variants of existing specific analyses. Examples of such non-amortized heap analyses for OO languages are in the recently appeared papers [6] and [9]. In the first paper the authors, given a set of constraints over input data, count the amount of paths which lead to calls of `new`. No recursive function calls are treated. The second paper is devoted to a type system, which allows to obtain symbolic expressions (where the free variables denote sizes of the inputs) that capture the amount of heap and stack memory required to execute the program. The type system treats aliasing as well.

Finally, it will be investigated how the new results that come up from the fundamental research line can be transferred to the more practical object-oriented setting.

6c. Relevance

Since memory exhaustion will invoke garbage collection, heap usage can indirectly slow down execution and hence influence time complexity. A better heap space analysis will therefore enable a more accurate estimation of time consumption. This is relevant for time-critical applications. Analysing resource usage is also interesting for optimisations in compilers for functional languages, in particular of memory allocation and garbage collection techniques. A more accurate estimation of heap usage enables allocation of larger memory chunks beforehand instead of allocating memory cells separately when needed, leading to a better cache performance.

Resource usage is an important aspect of any safety or security policy, as exhausting available resources typically causes system failure. Indeed, resource usage is one of the most important properties one wants to specify and verify for Java programs meant to be executed on (embedded) Java-enabled devices with limited amounts of memory, such as smart-cards implementing the Java Card platform, and MIDP mobile phones implementing the Java 2 Micro Edition (J2ME) platform.

The Java Programming Language (JML) already provides some rudimentary possibilities for specifying resource usage of Java programs. However, there is only syntax for specifying this, without any clear semantics, and there are no tools for actually monitoring – let alone, proving – that such constraints are met.

6d. Relationship with similar research

The most closely related research elsewhere is done at LMU (Ludwig Maximilians University) in Munich and LFCS (Laboratory for Foundation of Computer Science) in Edinburgh, especially in the MRG (Mobile Resource Guarantees) project supported

by the European Union (see http://groups.inf.ed.ax.uk/mrg). The work proposed here is in part complementary in that it considers lazy functional language. For the work on imperative language a difference is that we can build on a large existing body of work on Java verification.

The HIJA project (www.hija.org) is a project that studies the specification and verification of memory usage using the Java Modelling Language JML. This project focuses on the area of real-time applications.

Resource bounds are also studied in the ConCert project (Certified Code for Grid Computing, see http://www-2.cs.cmu.edu/~concert/) at CMU in the USA. But ConCert aims at producing proof carrying code, as does the MRG project, something not envisaged in this proposal.

We hope to be intensifying collaborations with LMU and LFCS in this project. We are currently in the process of trying to obtain (travel) funding at EU level for this. To cooperate with LMU, we may also try to obtain additional funding for travel through the cooperation program between NWO and DFG (Deutsche Forschungsgemeinschaft).

6e. Embedding

The computing science department at the Radboud University in Nijmegen provides an excellent environment for the proposed research, which spans functional and imperative languages. Van Eekelen and Smetsers contributed significantly to the programming language Clean and have expertise in functional languages and advanced type systems for functional languages. They recently joined the SoS group (formerly know as the LOOP group) of Jacobs and Poll. This group is one of the leading groups investigating semantics of Java and program verification. Considering that they complement each other very well we expect this project to lead to a productive synergy between these scientists.

The project also requires expertise in type theory, as the amortized analyses investigated use dependent types, and both Poll and Jacobs have a strong background in this field. Moreover, the department in Nijmegen also has the Foundations group renowned for its work on type theory.

7. Project Planning

Below we sketch a rough planning for the project. Of course, separation of work by the PhD student (4 years) and the post-doc (3 years) will not be as strict as indicated here.

Year 1: The PhD student gets acquainted with amortized analysis and the method of obtaining the linear bounds on heap space usage. For a simple functional language (strict core-Clean with lists as the only algebraic data type), (s)he then investigates type system(s) in which (dependent) types express limits on heap space usage. The PhD student studies example programs, finding out which dependencies arise from side conditions, and considers methods of finding solutions to these side conditions.

The post-doc gets acquainted with the type-theoretic memory model for LOOP [4], adapts this model to accurately reflect actual heap space usage of a Java Virtual Machine, and verifies heap space usage properties of some example Java

programs at the semantic level (i.e. by reasoning about the semantics of programs acting on an explicit representation of the heap). These results are compared with those in [5], which use a correspondence between Java programs and functional programs to obtain bounds on resource usage for the former.

Year 2+3: The PhD student investigates resource usage for lazy languages, by introducing lazy features into the programming language, and considers how the operational semantics and types system(s) can be adapted for this. In due course, extensions of the programming language with general algebraic data types rather than just lists, and improvements to analysis beyond linear bounds (as sketched in 6b.), will be tackled.

The post-doc develops the program logic for imperative languages, develops a prototype program logic for verification of JML-annotated Java programs. This program logic should be able to prove given resource bounds for simple programs. The restrictions for making the analysis applicable will be relaxed as much as possible. The program logic is integrated either with the LOOP tool or with ESC/Java2 [8], and tried out on serious examples. The post-doc also investigates possibilities to derive bounds, rather than prove given bounds, with as ultimate aim to implement a program that computes bounds.

Year 4: The PhD student will be using the fourth year for writing up.

Training of the PhD student

The PhD student employed in the project can take advantage of the local expertise in functional and object-oriented programming and type theory in Nijmegen, e.g. by participating in standard courses that are taught or activities organized on a more ad-hoc basis for PhD students. The PhD student will participate in the training program of the IPA research school and is also expected to participate in at least one international summer school.

8. Expected Use of Instrumentation

None.

9. Literature

Before summarising the references that are cited in this proposal we first give the five most important publications of the research team.

Five Most Important Publications

1. Bart Jacobs and Erik Poll, Coalgebras and Monads in the Semantics of Java. *Theoretical Computer Science*, 291(3):329–349, Elsevier, 2003.

2. L. Beringer and M. Hofmann and A. Momigliano and O. Shkaravska, *Automatic Certification of Heap Consumption*, in “Logic for Programming, Artificial Intelligence and Reasoning: 11th International Conference, LPAR 2004”. Vol. 3452 Springer-Verlag, 2005, pp. 347-367.

3. R. Plasmeijer, and M. van Eekelen, *Keep it Clean: A unique approach to functional programming*, In ACM Sigplan Notices, June 1999.

4. E. Barendsen, and S. Smetsers, *Uniqueness typing for functional languages with graph rewriting semantics*, In Mathematical Structures in Computer Science 6, pp. 579-612, 1996.

5. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, Springer Verlag, 2004. To appear.

References

[1] D. Aspinalll, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Proceedings of the International Workshop CASSIS 2004*.

[2] D. Aspinalll and M. Hofmann. Another type system for in-place update. In *ESOP'2002*, volume 2305 of *LNCS*, pages 36 – 52. Springer, 2002.

[3] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[4] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques (WADT'99)*, volume 1827 of *LNCS*. Springer, 2000.

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

[6] V. Braberman, D. Garbervetsky, and S. Yovine. Synthesizing parametric specifications of dynamic memory utilization in object-oriented programs. In *FTfJP 2005: Formal Techiques for Java-like Programs. Glasgow, Scotland*, July 2005.

[7] C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs. Proceedings of the ECOOP'2003 Workshop, Darmstadt, Germany*, pages 51–60, 2003. Technical Report 408, ETH Zurich.

[8] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. To appear. An earlier version appears in the proceedings of 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03), Volume 80 of ENTCS, Elsevier, 2003.

[9] W.-N. Chin, H. H. Nguen, S. Qin, and M. Rinard. Predictable memory usage for object-oriented programs. Technical report, National University of Singapore, Massachusetts Institute of Technology, 2004.

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

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

[12] B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. In *International Symposium on Software Security (ISSS'2003), Tokyo, Japan*, LNCS. Springer, 2004. To appear.

[13] S. Jost. From higher-order art to arthur. A Talk at the Colloquium of ”Graduiertenkolleg Logik in der Informatik” LMU Munich, December 2004.

[14] M. Konechny. Typing with conditions and guarantees for functional in-place update. In *TYPES 2002 Workshop, Nijmegen*, volume 2646 of *LNCS*, pages 182 – 199. Springer, 2003.

[15] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[16] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML reference manual. Available from `http://www.jmlspecs.org/Documentation`, 2002.

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

[18] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[19] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002. Invited Paper, LICS’02, 2002.29.

[20] Berry Schoenmakers. *Data Structures and Amostized Complexity in a Functional Setting*. PhD thesis, Eindhoven University of Technology, September 1992.

[21] Olha Shkaravska. Amortized heap-space analysis for first-order functional programs. Accepted at ”Trends in Functional Programming” TFP’05, Jule 2005.

[22] Olha Shkaravska. Types with semantics. Accepted at ”MEchanized Reasoning about Languages with varIable biNding” MERLIN’05, June 2005.

[23] M. van Eekelen and M. de Mol. Mixed lazy/strict natural semantics. Technical report, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, The Netherlands, 2004. Technical Report NIII-R0416.

10. Requested budget

In view of the breadth of the proposed research, which looks both at functional and imperative languages, support is requested for two positions, a three year post-doc and a PhD student. The post-doc will be focusing on transfer of results to imperative languages, which we feel requires more experience. For the post-doc position we have an excellent candidate, dr. Olha Shkaravska, currently working at Institute of Cybernetics, Tallinn.

Position	Salary	Bench fee
PhD student	160.029 EUR	5.000 EUR
Postdoc (3 yrs)	166.407 EUR	5.000 EUR
Totals	*326.436 EUR*	*10.000 EUR*

The total value of the application is *336.436 EURO*.